# MITIGATING CMP MEMORY WALL BY ACCURATE DATA PREFETCHING AND ON-CHIP STORAGE OPTIMIZATION

By

XUDONG SHI

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2007

To my wife and my parents

ACKNOWLEDGMENTS

I would thank my advisor, Dr. Jih-Kwon Peir, for his guidance and support throughout the whole period of my graduate study. His depth of knowledge, insightful advice, tremendous hard-working, great passion and persistence has been instrumental in the completion of this work. I thank Dr. Ye Xia for numerous discussions, suggestions and help on my latest research projects. I also extend my appreciation to my other committee members, Dr. Timothy Davis, Dr. Chris Jermaine and Dr. Kenneth O.

I appreciate the valuable help from my colleagues, Dr. Lu Peng, Zhen Yang, Feiqi Su, Li Chen, Sean Sun, Chung-Ching Peng, Zhuo Huang, Gang Liu, David Lin, Jianming Cheng, and Duckky Lee.

Finally, but most importantly, I would thank my parents and my wife for their endless love, understanding and support during my life. Without them, none of these would have been possible.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

# MITIGATING CMP MEMORY WALL BY ACCURATE DATA PREFETCHING AND ON-CHIP STORAGE OPTIMIZATION

By

Xudong Shi

December 2007

Chair: Jih-Kwon Peir
Major: Computer Engineering

Chip-Multiprocessors (CMPs) are becoming ubiquitous. With the processor feature size continuing to decrease, the number of cores in CMPs increases dramatically. To sustain the increasing chip-level power in many-core CMPs, tremendous pressures will be put on the memory hierarchy to supply instructions and data in a timely fashion. The dissertation develops several techniques to address the critical issues in bridging the CPU memory performance gap in CMPs.

An accurate, low-overhead data prefetching on CMPs has been proposed based on a unique observation of coterminous groups, highly repeated close-by off-chip memory accesses with equal reuse distances. The reuse distance of a memory reference is defined to be the number of distinct memory blocks between this memory reference and the previous reference to the same block. When a member in the coterminous group is accessed, the other members will likely be accessed in the near future. Coterminous groups are captured in a small table for accurate data prefetching. Performance evaluation demonstrates 10% IPC improvement for a wide variety of SPEC2000 workload mixes. It is appealing for future many-core CMPs due to its high accuracy and low overhead.

Optimizing limited on-chip cache space is essential for improving memory hierarchy performance. Accurate simulation of cache optimization for many-core CMPs is a challenge due to its complexity and simulation time. An analytical model is developed for fast estimating the performance of data replication in CMP caches. We also develop a single-pass global stack simulation for more detailed study of the tradeoff between the capacity and access latency in CMP caches. A wide-spectrum of the cache design space can be explored in a single simulation pass with high accuracy.

Maintaining cache coherence in future many-core CMPs presents difficult design challenges. The snooping-bus-based method and traditional directory protocols are not suitable for many-core CMPs. We investigate a new set-associative CMP coherence with small associativity, augmented with a Directory Lookaside Table (DLT) that allows blocks to be displaced from their primary sets for alleviating hot-set conflicts that cause unwanted block invalidations. Performance shows 6%-10% IPC improvement for both multiprogrammed and multithreaded workloads.

# CHAPTER 1
# INTRODUCTION

## 1.1 CMP Memory Wall

As the silicon VLSI integration continues to advance with deep submicron technology, billions of transistors will be available in a single processor die with a clock frequency approaching 10 GHz. Because of limited Instruction-Level Parallelism (ILP), design complexities, as well as high energy/power consumptions, further expanding wide-issued, out-of-order single-core processors with huge instruction windows and super-speculative execution techniques will suffer diminishing performance returns. It has become a norm that a processor die contains multiple cores, called a Chip Multiprocessor (CMP), and each core can execute multiple threads simultaneously to achieve a higher chip-level Instruction-Per-Cycle (IPC) [56]. The case for a chip multiprocessor was first presented in [56]. Since then, many companies have designed and/or announced their multi-core products [7], [40], [45], [39], [2], [35]. Trends, opportunities, and challenges for future CMPs have appeared in recent keynote speeches, invited talks, as well as in special columns of conferences and professional journals [15], [66], [69], [16]. CMPs are now becoming ubiquitous in all computing domains. As the processor feature size continues to decrease, the number of cores in a CMP increases rapidly. 4- or 8-core CMPs are now commercially available [3], [36], [29]. Recently, Intel announces a prototype of the teraflop processor [75], realizing an 80-core prototype with a 2D mesh interconnect architecture that reaches more than 1Tflops of performance. Furthermore, the advances of the wafer stacking technology, CAD design tools, thermal management, and electrothermal design methods make 3-dimentional (3D) chips feasible [48], [13]. This soon commercially available 3D technology further changes the landscape of the processor chips. We will see a large number of processor cores in a single CMP die, called many-core CMPs.

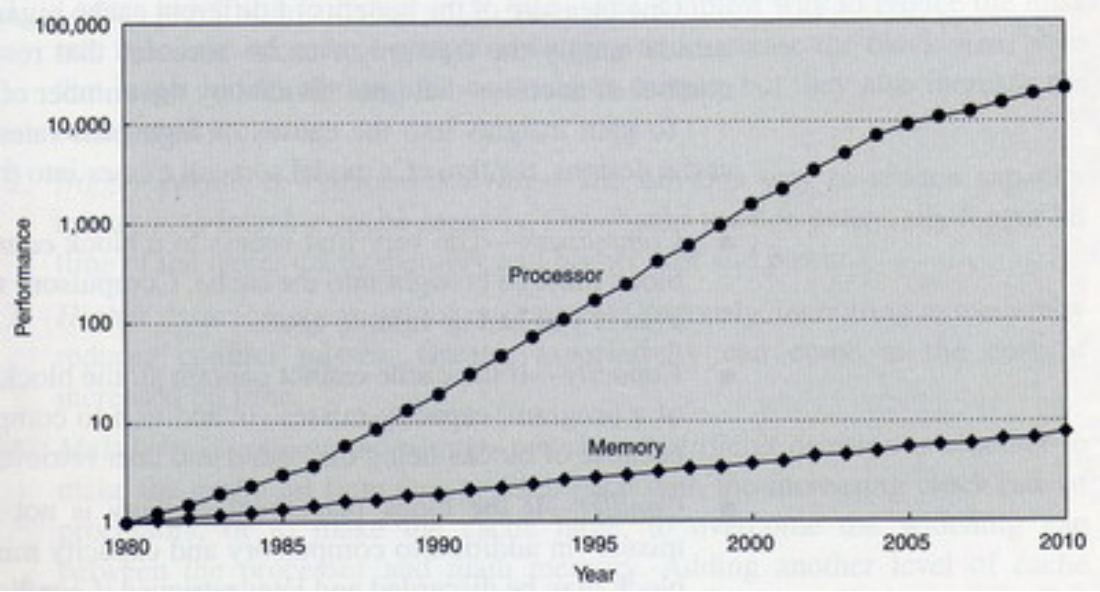

Figure 1-1. Performance gap between memory and cores since 1980. (J.Hennessy and D.Patterson, Computer Architecture: A Quantitative Approach 4th Edition).

In such a die with a large number of cores, tremendous pressure will be put on the memory hierarchy system to supply the needed instructions and data in a timely fashion to sustain ever-increasing chip-level IPCs. However, the performance gap between memory and cores has been widening since 1980, as illustrated in Figure 1-1. It becomes a critical question of how to bridge the performance gap between processors and memory on many-core CMPs.

Hierarchical caches are traditionally designed to bridge the gap between main memory and cores by utilizing programs' *spatial locality* and *temporal locality*. To match the fast speed and high bandwidth of the CPU's execution pipelines, it is quite a standard that every core in CMPs usually has small (e.g. 16KB - 64KB) first level private instruction and data caches, tightly coupled into its pipelines. The most critical part of the many-core CMP cache hierarchy design boils down to the lower level caches. However, several challenges exist to design the lower level caches in many-core CMPs. First, since caches occupy a large percentage of die space, the total cache space in many-core CMPs is usually restricted in order to keep the die footprint reasonably sized. It is even not unusual that average cache space per core decreases when the number of

cores increases. How to leverage the precious on-chip storage space becomes an essential issue. Second, many-core CMPs usually suffer from longer on-chip remote cache access latency and off-chip memory access latency in terms of number of CPU cycles. This is mainly due to two reasons. On one hand, the gap between the speed of cores and that of caches/memory is widening. On the other hand, with large number of cores and large amount of cache space, it is extremely time-consuming to locate and transfer the data block in/between the caches. A critical question is how to reduce the on-chip and off-chip access latency. Third, many-core CMPs demands higher on- and off-chip memory bandwidth. To sustain the IPC of many cores, large amount of data must be transferred between main memory and caches, among different caches and between caches and cores.

## 1.2 Directions and Related Works

The design of the lower level caches in the next-generation many-core CMPs has yet to be standardized. There are many factors to leverage and many possible performance metrics to evaluate. Several very important directions among them are data prefetching, optimizing the on-chip cache organization and improving cache coherence activities among different on-die caches. In the following part of this chapter, we will give an introduction of existing solutions in these directions and raise some interesting questions.

### 1.2.1 Data Prefetching

Data prefetching has been an important technique to mitigate the CPU and memory performance gap [38], [22], [37], [74], [77], [33], [54], [46]. It speculatively fetches the data or instructions that the processor(s) will likely use in the near future into the on-chip caches in advance. A successful prefetch may change an off-chip memory miss into a cache hit or a partial cache hit, thus eliminate or shorten the expensive off-chip memory latency. A data prefetcher may make decisions on what and when to prefetch based on program semantics that have been

hinted by programmers or compilers (software-based) or based on runtime memory access patterns that the processors have experienced (hardware-based). The software techniques require a significant knowledge and effort from programmers and/or compilers, which limits their practicability. The hardware approaches, on the other hand, predict the future access pattern based on the history, and hope that the history will reoccur in the future. However, an inaccurate (useless) prefetch may hurt the overall performance since the useless prefetch consume memory bandwidth, and the useless data block may pollute the on-chip storage.

Three key metrics are used to measure the effectiveness of a data prefetcher [37]:

**Acuracy**. Accuracy is defined as the ratio of useful prefetches to total prefetches.

**Coverage**. Coverage is the ratio of useful prefetches to total number of misses. It is the percentage of misses that have been covered by prefetching.

**History size**. History size is the size of the extra history table used to store memory access patterns, usually for hardware prefetchers.

Many uni-processor data prefetching schemes have been proposed in the last decade [37], [74], [77], [54]. Traditional sequential or stride prefetchers identify sequential or stride memory access pattern, and prefetch next a few blocks in such a pattern. They work well for workloads with regular spatial access behaviors [38], [22], [46]. Correlation-based predictors (e.g. Markov predictor [37] and Global History Buffer [54]) record and use past miss correlations to predict future cache misses. They record miss pairs of (A->B) in a history table, meaning that a miss B following the miss A has been observed in the past. When A misses again, B will be prefetched. However, a huge history table or a FIFO buffer is usually necessary to provide decent coverage. Instead of recording individual miss block correlations, Hu et al. [33] uses tag-correlation, a much bigger block correlation, to reduce the history size. The down side of the bigger block

correction is that it reduces the accuracy as well. To avoid cache pollution and provide timely prefetches, the dead-block prefetcher issues a prefetch once a cache block is predicted to be dead, based on a huge history of program instructions [47].

Speculative data prefetching become more essential and more necessitating on CMPs to hide the higher memory wall. But prefetching on CMPs is more challenging due to limited on-die cache space and off-chip memory bandwidth. Traditional Markov data prefetcher [37], despite its advantage of reasonable coverage and great generality, faces serious obstacles in the context of CMPs. First, each cache miss often has several potential successive misses and prefetching multiple successors is inaccurate and expensive. Such incorrect speculations are more harmful on CMPs, wasting already limited memory bandwidth and polluting critical on-chip caches. Second, consecutive cache misses can be separated by few instructions. It could be too late to initiate prefetches for successive misses. Third, reasonable miss coverage requires a large history table which translates to more on-chip power/area.

Recently, several proposals target to improve prefetch accuracy. Saulsbury et al. [63] proposed a recency-based TLB preloading. It maintains the TLB information in a Mattson stack, and preloads adjacent entries in the stack upon a TLB miss. The recency-based technique can be applied for data prefetching. However, it prefetches adjacent entries in the stack without the prior knowledge of whether the adjacent requests have showed any repeated patterns or how the two requests arrive at the adjacent stack positions. Chilimbi [23] introduced a hot-stream prefetcher. It profiles and analyzes sampled memory traces on-line to identify frequently repeated sequences (hot streams) and inserts prefetching instructions to the binary code for these streams. The profiling, analysis, and binary code insertions / modifications incur execution overheads, and may become excessive to cover hot streams with long reuse distances. Wenisch et al. [78]

proposed temporal streams by extending hot streams and global history buffer to deal with coherence misses on SMPs. It requires a huge FIFO and multiple searches/comparisons on every miss to capture repeated streams.

In spite of so much effect, it remains a big challenge to provide a more accurate data prefetcher with low overhead on CMPs, where memory bandwidth and chip space are more limited, and where inaccurate prefetches are less tolerated.

### 1.2.2 Optimization of On-Chip Cache Organization

With limited on-chip caches, optimization of on-chip lower level cache organization becomes critical. An important design metrics is whether the lower level cache is shared among many cores or partitioned into private caches for each core. Sharing has two main benefits. First, sharing increases the effective capacity of the cache, since a block only has one copy in the shared cache. Second, sharing balances cache occupancy automatically among workloads with unbalanced working sets. However, sharing often increases the hit latency due to the longer wiring delay, and possibly also due to larger search time and bandwidth bottleneck. Furthermore, a dynamic sharing may lead to erratic, application-dependent performance when different cores interfere with each other by evicting each other's blocks. It causes priority-inversion problem when the task running in one core occupies too much cache space and starves higher priority tasks in other cores [42], [61].

A monolithic shared cache with high associativity consumes more power as the size increases. Non-uniform cache access (NUCA) [41] architecture splits a large monolithic shared cache into several banks to reduce power dissipation and increase bandwidth. Usually the number of banks is equal to the number of cores, and each core has a local bank. Which bank to store a block is statically decided by the lower bits of block addresses. The access latency thus depends on the distance between the requesting core and the bank containing the data. Generally,

only a small fraction of accesses (the reciprocal of the number of cores) are targeting to local banks.

Alternatively, private caches contain most recently used blocks for specific cores. They provide fast local accesses for majority of the cache hits, probably reducing the traffic between different caches and consuming less power. But, data may be replicated in private caches when two or more cores share the same blocks, leading to less capacity and often more off-chip memory misses. Private caches also need to maintain data coherence. Upon a local read miss or a write without data exclusivity, the accessing core needs to check other private caches by either a broadcast or through a global directory, to fetch data and/or to maintain write consistency by invalidating remote copies. Another downside is private caches do not allow storage space sharing among multiple cores, thus can not accommodate unbalanced cache occupancy for workloads with different working sets.

It has become increasingly clear that it could be better to combine the benefits of both private caches' and shared caches' [49], [24], [81], [34], [20], [9]. Generally, they can be summarized into two general directions. The first direction is to organize the L2 as a shared cache for maximizing the capacity. To shorten the access time, data blocks may be dynamically migrated to the requesting cores [8], and/or some degree of replication is allowed[81], [9], to increase the number of local accesses at a minimum cost of lowering on-chip capacity. To achieve fair capacity sharing, [60] partitions a shared cache between multiple applications depending on the reduction in cache misses that each application is likely to obtain for a given amount of cache resources. The second direction is to organize the L2 as private caches for minimizing the access time. But data replications among multiple L2s are constrained to achieve larger effective on-chip capacity [20] without adversely decreasing the number of local accesses

too much. Dybdahl [26] proposes to create a shared logical L3 part by giving up a dynamically adjusted portion of private L2 space for each core. To achieve optimal capacity sharing, private L2s can steal other's capacity by block migration [24], [20], accommodating different space requirements of different cores (workloads).

One of the biggest problems that these studies face is extremely long simulation time. They must examine a wide-spectrum of design spaces to have complete conclusions, such as different number of cores, different L2 sizes, different L2 organizations, and different workloads with different working sets. Furthermore, increasing number of cores on CMPs makes the problem even worse. Simulation time usually increases more than linearly as the number of cores increases. It is expected that 32, 64 or even hundreds of cores will be the target of the future research. To reduce the simulation time, FPGA simulation might be a good solution, but it is too difficult to build. A great challenge is then how to provide an efficient methodology to study design choices of optimizing CMP on-chip storage accurately and completely, when the number of cores increases.

There have been several techniques for speeding up cache simulations in uniprocessor systems. Mattson, et al. [53] presents a LRU stack algorithm to measure cache misses for multiple cache sizes in a single pass. For fast search through the stack, tree-based stack algorithms are proposed [10], [76]. Kim et al. [43] provide a much faster simulation by maintaining the reuse distance counts only to a few potential cache sizes. All-associativity simulations allow a single-pass simulation for variable set-associativities [32], [72]. Meanwhile, various prediction models have been proposed to provide quick cache performance estimation [5], [28], [27], [76], [11], [12]. They apply statistical models to analyze the stack reuse distances. But, it is generally difficult to model systems with complex real-time interactions among

multiple processors. StatCache [11] estimates capacity misses using sparse sampling and static statistical analysis.

All above techniques target uniprocessor systems where there is no interference between multiple threads. Several works aim at modeling multiprocessor systems [79], [80], [19], [12]. StatCacheMP [12] extends StatCache to incorporate communication misses. However, it assumes a random replacement policy for the statistical model. Chandra, et al [19] propose three analytical models based on the L2 stack distance or circular sequence profile of each thread to predict inter-thread cache contentions on the CMP for multiprogrammed workloads that do not have interference with each other. Two other works pay attention only to miss ratios, update ratios, and invalidate ratios for multiprocessor caches [79], [80].

Despite those efforts, it is still an important problem how to efficiently model and predict the performance results of optimization of many-core CMP cache organization about the tradeoff between data capacity and accessibility.

### 1.2.3 Maintaining On-Die Cache Coherence

Cache Coherence defines the behavior of reads and writes to the same memory location with multiple cores and multiple caches. The coherence of caches is obtained if the following conditions are met: 1) Program order must be preserved among reads and writes from the same core. 2) The coherent view of memory must be maintained, i.e. a read from a core must return the latest value written by other cores. 3) Writes to the same location must be serialized.

On CMPs, since the on-chip cache access latency is much less than the off-chip memory latency, it is desirable to obtain the data from on-chip caches if possible. Write-invalidation cache coherence protocol is generally used on today's microprocessors. There are three cache coherence activities here on CMPs. First, on a data read miss at the local cache, a search through the other on-chip caches is performed to obtain the latest data, if possible. Second, on a write

miss at the local cache, a search through the other caches is performed to obtain the latest data if possible and all those copies must be invalidated. Third, on a write upgrade (i.e. write hit at the local cache without exclusivity), all the copies at other caches must be invalidated.

Maintaining cache coherence with increasing number of cores has become a key design issue for future CMPs. In an architecture with a large number of cores and cache modules, it becomes inherently difficult to locate a copy (or copies) of a requested data block and to keep them coherent. When a requested block is not located in the local cache module, one strategy is to search through all modules. A broadcast to all the modules is possible at the same time or in the sequence of multiple levels, for instance, local modules, neighbor modules, and entire space. This approach is only applicable to a system with a small number of cores using a shared snooping bus. Searching the entire CMP cache becomes prohibitively time consuming and power hungry when the number of cores increases. Hierarchical clusters with hierarchical buses alleviate the problem at the expense of introducing lots of complexity. Recent ring based architecture connects all the cores (and the local L2 slice) with one or more uni- or bi-directional rings [44], [71]. A remote request travels hop-by-hop on the ring until a hit is encountered. The data may travel back to the requesting core hop-by-hop or directly depending on whether data interconnection shortcuts are provided. The total number of hops varies, depending on the workloads and the data replication strategy. However, ring-based architecture may still not scale well as the number of cores increases.

To avoid broadcasting and searching all cache modules, directory-based cache-coherence mechanisms might be the choice. When a request cannot be serviced by local cache module, this request is sent to the directory to find the state and the locations of the block. Many directory implementations have been proposed in the field of Symmetric Multiprocessor (SMP). A

memory-based directory records the states and sharers of all memory blocks at each memory module using a set of presence bits [17]. Although the memory-based directory can be accessed directly by the memory address, such a full directory is unnecessary in CMPs since the size of the cache is only a small fraction of the total memory. There have been many research works trying to overcome the space overhead of the memory-based directory [4], [18], [25]. Recently, a multi-level directory combines the full memory-based directory with directory caches for fast accesses [1]. A directory cache is a small full-map, first-level directory that provides information for the most recently referenced blocks, while the second-level directory provides additional information for all the blocks. The cache-based directory, on the other hand, records only cached blocks in the directory to save directory space. The simplest approach is to duplicate all individual cache directories in a centralized location [73]. For instance, Piranha [7] duplicates L1 tag arrays in the shared L2 to maintain L1 coherence. Searches to all cache directories are necessary to locate copies of a block. This essentially builds a directory of much wide set-associability (the multiplication of number of cores and number of cache ways per set), and wastes a lot of power. In a recent virtual hierarchy design [51], a 2-level directory is maintained in a virtual machine (VM) environment. The level-1 coherence directory is embedded in the L2 tag array of the dynamically mapped home tile located within each VM domain. Any unresolved accesses will be sent to the level-2 directory. If a block is predicted on-chip, the request is broadcast to all cores.

The sparse directory approach uses a small fraction of the full memory directory organized in a set-associative fashion to record only those cached memory blocks [30], [55]. Since directory must maintain the full map of cache states, hot-set conflicts at the directory lead to unnecessary block invalidations at the cache modules, resulting in an increase of cache miss.

With a typical set-associative directory, such conflict at the directory tends to become worst as the number of cores increases, unless the set-associativity also dramatically increases. For instance, in a CMP with 64 cores with each core having a 16-way local cache module, only a 1024-way directory can eliminate all inadvertent invalidations. A naive plan of building a 1024-way set-associative directory, although it can eliminate all conflicts, it is hardly feasible. Thus, an important technical problem is to avoid the hot-set conflicts at the directory with small set associativity, small space and high efficiency.

Some previous works exist to alleviate the hot-set conflict of caches, instead of the cache coherence directory. The column-associative cache [6] establishes a secondary set for each block using a different hash function from that for the primary set. The group-associative cache [58] maintains a separate cache tag array for a more flexible secondary set. The skewed-associative cache applies different hash functions on different cache partitions to reduce conflicts [64], [14]. The V-way cache [59] doubles the cache tag size with respect to the actual number of blocks to alleviate set conflicts. Any unused tag entry in the primary set can record a newly missed block without replacing an existing block. The Bloomier filter [21] approach institutes a conflict-free mapping from a set of search keys to a linear array, using multiple hash functions. It remains a problem to reduce the set-conflict of cache coherence directory.

### 1.2.4 General Organization of Many-Core CMP Platform

To establish the foundation for the proposed research, we plot the possible organization of future many-core CMPs, as illustrated in Figure 1-2. This is similar to the Intel's vision of future CMPs. The CMPs will be built on a partition-based or tiled-based substrate. There will be 16-64 processing cores and 16-64MB on-chip cache capacity.

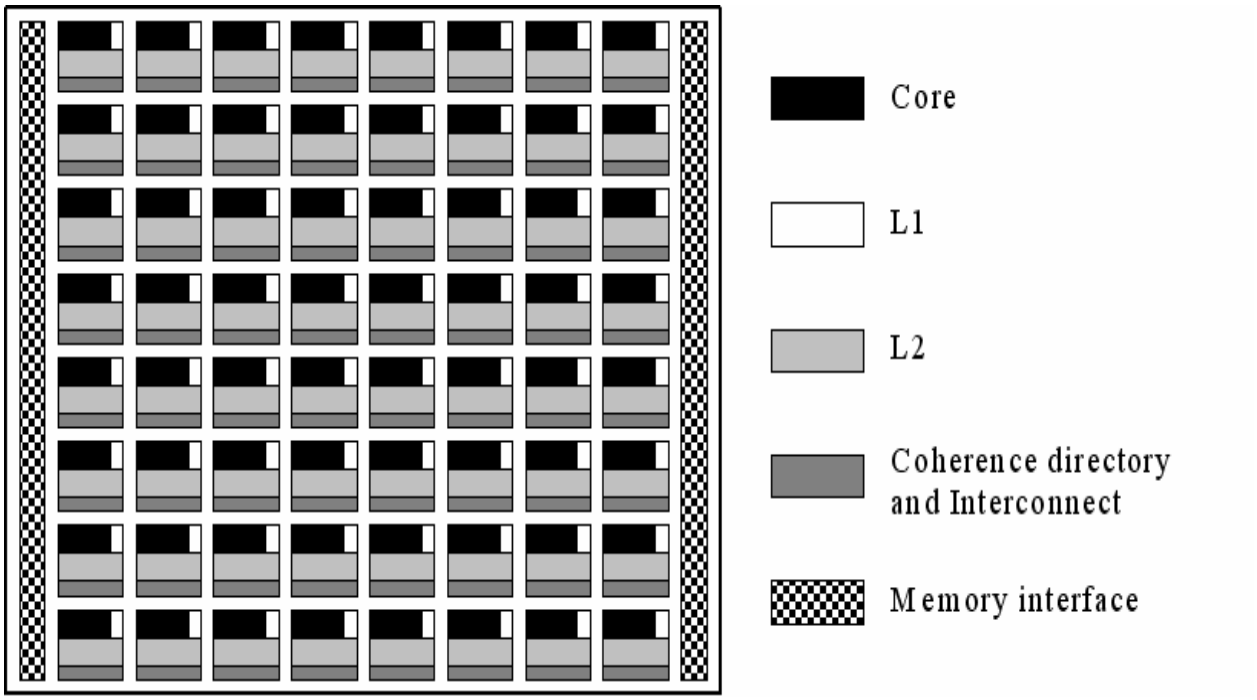

Figure 1-2. Possible organization of the next-generation CMP.

Each tile contains one core and a local cache partition. To match the speed and bandwidth of processor internal pipelines, the local cache partition is further divided into even closer private L1 instruction and data caches, and a unified local L2 module. All the other L2 modules except the local module are remote to this specific core. For achieving the shortest access time, each memory block may be dynamically allocated, migrated, and replicated to any individual L2 cache modules.

The coherence directory is also physically distributed among all the partitions, though logically it is centralized. Write-invalidate write-allocate, MOESI coherence protocol is applied to maintain data coherence for blocks allocated to multiple cache modules. The coherence directory maintains the states of all cache blocks. A directory lookup request will be sent to the directory if a memory request cannot be honored by the local cache, such as read miss, write miss, and write upgrade. It is likely that a 2D mesh interconnect and necessary routers link all the partitions. The access latency to a remote partition is decided by the wiring distance and router

processing time. Multiple memory interfaces may be available to support multiple channel main memories.

## 1.3 Dissertation Contribution

There are three main contributions in this dissertation addressing the issue of mitigating the CMP memory wall. First of all, we develop an accurate and low overhead data prefetching based on a unique observation of the program behavior. Second, we describe an analytical model and a single-pass global stack simulation to fast project the performance of the tradeoff between cache capacity and data accessibility in CMP on-chip caches. Third, we develop a many-core CMP coherence directory with small set associativity, small space and high efficiency and introduce a directory lookaside table to reduce the number of inadvertent cache invalidations due to directory hot-set conflicts.

### 1.3.1 Coterminous Group Data Prefetching

- We prove with a set of SPEC CPU 2000 workload mixes that cache contentions from different cores running independent workloads creates much more cache misses.
- We observer a unique behavior of *Coterminous Group* (CG) in the SPECCPU 2000 applications, a group of memory accesses with temporal repeated memory access patterns. We further define a new *Coterminous Locality*: when a member in a CG is referenced, the other members will be reference soon.
- We develop a CG-prefetcher based on Coterminous Groups. It identifies and records highly repeated CGs in a small buffer for accurate and timely prefetches for members in a group. Detailed performance evaluations have shown significant IPC improvement against other prefetchers.

### 1.3.2 Performance Projection of On-chip Storage Optimization

- We present an analytical model for fast projection of CMP cache performance about the tradeoff between data accessibility and the cache capacity loss due to data replication.
- We develop a single-pass global stack simulation to more accurately simulate these effects for shared and private caches. By using the stack results of the shared and private caches, we further deduce the performance effect of more complicated cache organizations, such as shared caches with replication and private caches without replication.

- We verify the projection accuracy of the analytical model and the stack simulation by detailed execution-driven simulation. More importantly, the single-pass global stack simulation only consumes a small percentage of simulation time that execution-driven simulation requires.

### 1.3.3 Enabling Scalable and Low-Conflict CMP Coherence Directory

- We demonstrate the sparse coherence directory with small associativity causes significant unwanted cache invalidation due to the hot-set conflicts in the coherence directory.

- We augment a *Directory Lookaside Table* (DLT) for the set-associative sparse coherence directory to allow the displacement of a block away from its primary set to one of the empty slots in order to reduce conflict.

- Performance evaluations using multithreaded and multiprogrammed workloads demonstrate significant performance improvement of the DLT-enhanced directory over the traditional set-associative or skewed associative directories by eliminating majority of the inadvertent cache invalidations.

## 1.4 Simulation Methodology and Workload

To implement and verify our ideas, we use the full-system simulator, Virtutech Simics 2.2 [50], to simulate 2-, 4-, or 8-core CMPs running real operation system Linux 9.0 on a machine with x86 Instruction Set Architecture (ISA).

The processor module is based on the Simics Microarchitecture Interface (MAI) and models timing-directed processors in detail. A g-share branch predictor is added to each core. Each core has its own instruction and data L1 cache. Since L2 cache is our focus, we have different L2 organizations in different works. It will be described in each work later.

We implement a cycle-by-cycle event-driven memory simulator to accurately model the memory system. Multi-channel DDR SDRAM is simulated. The DRAM accesses are pipelined whenever possible. A cycle-accurate, split-transaction processor- memory bus is also included to connect the L2 caches and the main memory.

Table 1-1 summarizes common simulation parameters. Parameters that are specific to each work will be described later in the individual chapters.

Table 1-1. Common simulation parameters

| Parameter | Description |
|---|---|
| CMP | 2, 4, or 8 cores, 3.2GHz, 128-entry ROB |
| Pipeline Width | 4 Fetch / 7 Exec / 5 Retire / 3 Commit |
| Branch predictor | G-share, 64KB, 4K BTB, 10 cycle misprediction penalty |
| L1-I/L1-D | 64KB, 4-way, 64B Line, 16-entry MSHR, MESI, 0/2-cycle latency |
| L2 | 8- or 16-way, 64B Line, 16-entry MSHR, MOESI if not pure shared |
| L2 latency | 15 cycles local, 30 cycles remote |
| Memory latency | 432 cycles without contentions |
| DRAM | 2/4/8/16 channels, 180-cycle access latency |
| Memory bus | 8-byte, 800MHz, 6.4GB/s, 220-cycle round trip latency |

We use 2 sets of workloads in our study: multiprogrammed and multithreaded workloads. For multiprogrammed workloads, we use several mixtures of SPEC CPU2000 and SPEC CPU2006 benchmark applications based on the classification of memory-bound and CPU-bound applications [82]. The memory-bound applications are Art, Mcf, and Ammp, while the CPU-bound applications are Twolf, Parser, Vortex, Bzip2 and Crafty. The first category of workload mixes, MEM, includes memory-bound applications; the second category of workload mixes, MIX, consists of both memory-bound and CPU-bound applications; and the third category of workloads, CPU, contains only CPU-bound applications. For studies with 2-, 4-, or 8-core CMPs, we prepare 2-, 4- and 8-application workloads. We choose the ref input set for all the SPEC CPU2000 and SPEC CPU2006 applications.

Table 1-2 summarizes the selected multiprogrammed workload mixes. For each workload mix, the applications are ordered by high-to-low L2 miss penalties from left to right in their appearance. We skip certain instructions for each individual application in a mix based on studies done in [62], and run the workload mix for another 100 million instructions for warming up the caches. A Simics checkpoint for each mix is generated afterwards. We run our simulator until any application in a mix has executed at least 100 million instructions for collecting statistics.

Table 1-2. Multiprogrammed workload mixes simulated

| | MEM | MIX | CPU |
|---|---|---|---|
| Two | Art/Mcf<br>Mcf/Mcf<br>Mcf/Ammp | Art/Twolf<br>Mcf/Twolf<br>Mcf/Bzip2 | Twolf/Bzip2<br>Parser/Bzip2<br>Bzip2/Bzip2 |
| Four | Art/Mcf/Ammp/Twolf | Art/Mcf/Vortex/Bzip2 | Twolf/Parser/Vortex/Bzip2 |
| Eight | Art/Mcf/Ammp/Parser/Vortex/Bzip2/Crafty (SPEC2000)<br>Mcf/Libquantum/Astar/Gobmk/Sjeng/Xalan/Bzip2/Gcc (SPEC2006) | | |

We also use three multithreaded commercial workloads, OLTP (Online Transaction Processing), Apache (Static web server), and SPECjbb (java server). We consider the variability of these multithreaded workloads by running multiple simulations for each configuration of each workload and inserting small random noises (perturbations) in the memory system timing for each run. For each workload, we carefully adjust system and workload parameters to keep the CPU idle time low enough. We then fast-forward the whole system for enough period of time to fill the internal buffers and other structures before making a checkpoint. Finally, we collect simulation results during executing a certain number of transactions after we warm up the caches or other simulation related structures. Table 1-3 gives the details of the workloads.

## 1.5 Dissertation Structure

The structure of this dissertation is as follows. Chapter 2 develops the first piece of the dissertation, an accurate and low-overhead data prefetching technique in CMPs based on a unique observation of coterminous group, a highly repeated and close-by memory access sequence. In Chapter 3, we illustrate two methodologies, an abstract data replication model and a single-pass global stack simulation, to fast project the performance issue of CMP on-chip storage optimization. Chapter 4 builds a set-associative CMP coherence directory with small associativity and high efficiency, augmented by a directory lookaside table that alleviates the directory hot-set conflicts. This is followed by a brief summary of the dissertation in Chapter 5.

Table 1-3. Multithreaded workloads simulated

| Workload | Description |
|---|---|
| OLTP (Online Transaction Processing) | It is built upon the OSDL-DBT-2 [57] and the MySQL database server 5.0. We build a 1GB, 10-warehouse database. To reduce the database disk activity, we increase the size of the MySQL buffer pool to 512MB. We further stress the system by simulating 128 users without any keying and thinking time. We simulate 1024 transactions after bypassing 2000 transactions and warming up caches (or stack) for another 256 transactions. |
| Apache (Static web server) | We run apache 2.2 from as the web server, and use Surge to generate web requests from a 10,000 file, about 200MB repository. To stress the CMP system, we simulate 8 clients with 50 threads per client. We collect statistics for 8192 transactions after bypassing 2500 requests and warming up for 2048 transactions. |
| SPECjbb (java server) | SPECjbb is a java-based 3-tier online transaction processing system. We simulate 8 warehouses. We first fast-forward 100,000 transactions. Then we simulate 20480 transactions after warming up the structures for 4096 transactions. |

# CHAPTER 2
# COTERMINOUS GROUP DATA PREFETCHING

In this chapter, we describe an accurate data prefetching technique on CMPs to overlap expensive off-chip cache miss latency. Our analysis of SPEC applications shows that adjacent traversals of various data structures, such as arrays, trees and graphs, often exhibit temporal repeated memory access patterns. A unique feature of these nearby accesses is that they exhibit a long but equal reuse distance. The *reuse distance* of a memory reference is defined as the number of distinct data blocks that are accessed between this reference and the last references to the same block. It is the most fundamental measure of memory reference locality. We define such a group of memory references with an equal block reuse distance as a *Coterminous Group* (CG) and the highly repeated access patterns among members in a CG as the *Coterminous Locality*. A new data prefetcher identifies and records highly repeated CGs in a history buffer. For accurate and timely prefetches, whenever a member in a CG is referenced, the entire group members are prefetched. We call such a data prefetching method a *CG-prefetcher*.

We make three contributions about the CG-prefetcher. First, we demonstrate the severe cache contention problem with various mixes of SPEC2000 applications, and describe the necessities and the challenges of accurate data prefetching on CMPs. Second, we discover the existence of coterminous groups in these applications and quantify the strong coterminous locality among members in a CG. Third, based on the concept of coterminous groups, we develop a new CG-prefetcher, and present a realistic implementation by integrating the CG-prefetcher into the memory controller. Full system evaluations based on mixed SPEC CPU 2000 applications have shown that the proposed CG-prefetcher can accurately prefetch the needed data in a timely manner on CMPs. It generates about 10-40% extra traffic to achieve 20-50% of miss coverage in comparison with two and a half times more extra traffic by a typical correlation-

based prefetcher with a comparable miss coverage. The CG-prefetcher also shows better IPC (Instructions per Cycle) improvement than the existing miss correlation based or the stream based prefetchers.

To clearly demonstrate the effectiveness of CG-prefetcher, we carry out experiments on a simple shared L2 cache with multiple cores, each running a different application. But, this scheme is independent of any specific cache organizations, and can be adapted to private caches as well.

### 2.1 Cache Contentions on CMPs

CMPs put tremendously more pressure on the memory hierarchy to supply the data to cores than uniprocessors. One of the major reasons is that different cores compete for the limited on-chip shared storage when multiple independent applications are running simultaneously on multiple cores. The typical example is the shared L2 cache among multiple cores. This effect is more evident when independent memory-intensive applications are running together. To demonstrate the cache contentions on CMPs, we show the IPCs (Instructions per Cycle) of a set of SPEC2000 applications that are running individually, or in parallel on 4- or 2-core CMPs in Figure 2-1 (A) and Figure 2-1 (B) respectively.

We have three workload mixes consisting of 4 applications, Art/Mcf/Ammp/Twolf, Art/Mcf/Vortex/Bzip2, and Twolf/Parser/Vortex/Bzip2 in Figure 2-1 (A). The first workload mix, Art/Mcf/Ammp/Twolf, contains applications with heavier L2 misses; the second one, Art/Mcf/Vortex/Bzip2, mixes applications with heavier and lighter L2 penalties; and the third one, Twolf/Parser/Vortex/Bzip2, has applications with generally lighter L2 misses. We also run nine 2-application mixes in Figure 2-1 (B), also ranging from high-to-low L2 miss penalties, including Art/Mcf, Mcf/Mcf, Mcf/Ammp, Art/Twolf, Mcf/Twolf, Mcf/Bzip2, Twolf/Bzip2, Parser/Bzip2, and Bzip2/Bzip2.

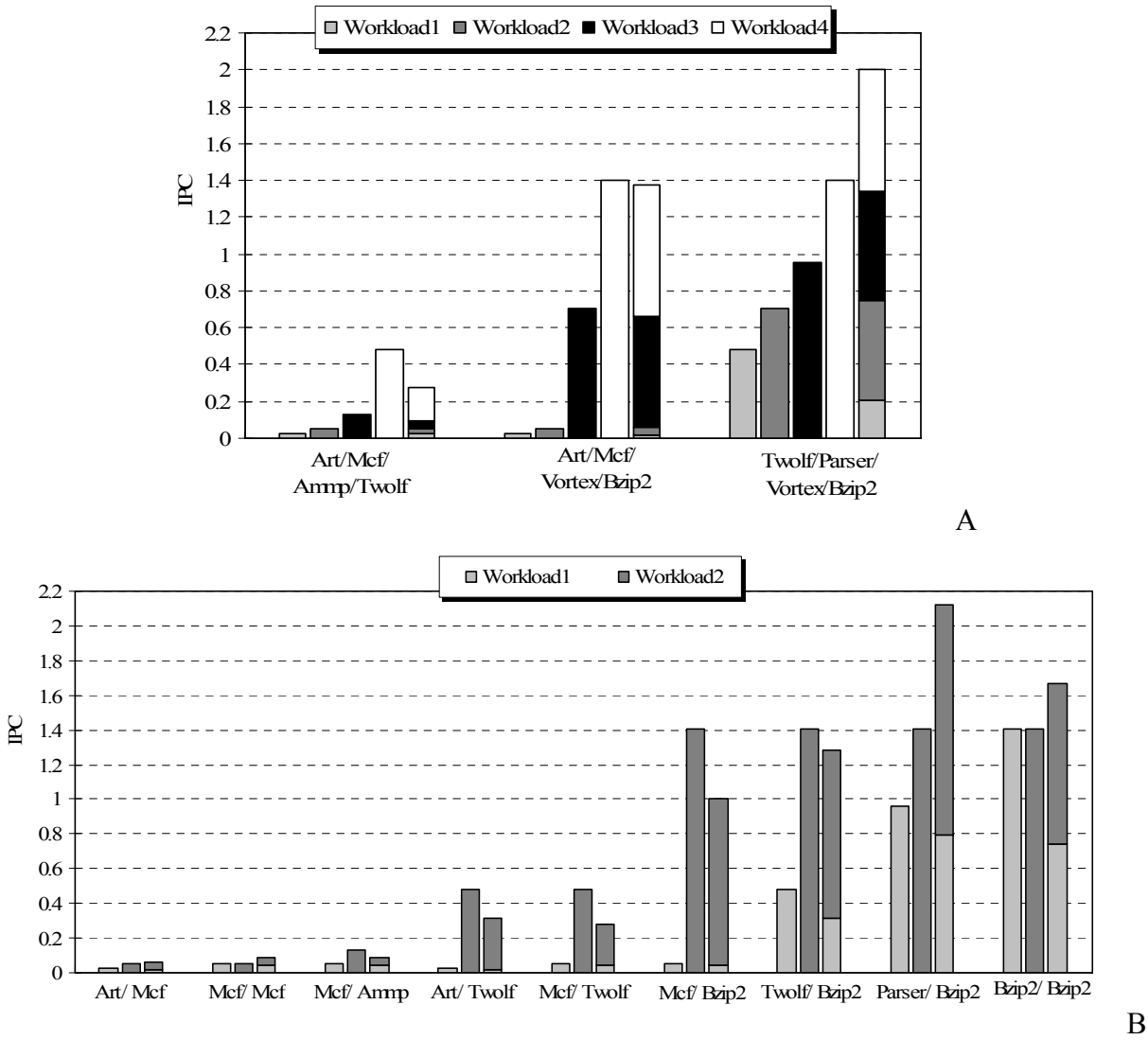


Figure 2-1. IPC degradation due to cache contention for SPEC2000 workload mixes on CMPs. A) IPC for 4-workload on 4-core CMPs. The first 4 bars are individual IPCs when only one application is running and the last bar is combined IPC when 4 workloads run in parallel on 4 cores. B) IPC for 2-workload on 2-core CMPs. The first 2 bars are individual IPCs when only one application is running and the last bar is combined IPC when 2 workloads run in parallel on 2 cores.

The first four bars in Figure 2-1 (A) and the first two bars in Figure 2-1 (B) are the individual IPCs (Instructions per Cycle) of the applications in the workload mixes, ordered by the appearance in the name of the workload mixes. We collect the individual IPC for each application by running the specific application on one core and keeping all the other cores idle. As a result, the entire L2 cache is available for the individual application. The last bar of each

workload mix is the combined IPC when we run all the applications at the same time with one core running one independent application. The combined IPC is broken down into segments to show the IPC contribution of each application.

Ideally, the combined IPC should be equal to the sum of individual IPCs when only one application is running. However, significant IPC reductions can be observed on each application when they run in parallel, mainly due to the shared L2 cache contention among multiple applications. This is especially evident for the workload mixes with high demands on shared L2 caches. For example, when Art/Mcf/Ammp/Twolf are running on four cores, the individual IPC drops from 0.029 to 0.022 for Art, from 0.050 to 0.026 for Mcf, from 0.132 to 0.043 for Ammp, and from 0.481 to 0.181 for Twolf respectively. Instead of accumulating the individual IPCs on four cores, the combined IPC drops from 0.69 (the sum of individual IPCs) to 0.27, a 60% of degradation. Similar effects of various degrees can also be observed with 2-core CMPs. The significant IPC degradations come from more L2 misses and more off-chip memory accesses.

Data prefetching is an effective way to reduce number of L2 misses. However, in the CMP context with limited cache space and limited memory bandwidth, inaccurate prefetches are more harmful. Those inaccurate prefetches pollute the limited cache space and wasted the limited memory bandwidth. The CMP demands for accurate prefetchers with low overhead to alleviate heavier cache contentions and misses.

## 2.2 Coterminous Group and Locality

The proposed data prefetcher on CMPs is based on a unique observation of the existence of *Coterminous Groups (CGs)*. A *Coterminous Group (CG)* is a group of nearby data references with *same block reuse distances*. The *reuse distance* of a reference is defined as the number of **distinct** data blocks that are accessed between two consecutive references to this block. For instance, consider the following accessing sequence: ***a b c*** *x* ***d*** *x y z* ***a b c*** *y* ***d***. The reuse distances

of *a*-*a*, *b*-*b*, *c*-*c* and *d*-*d* are all 6, whereas *x*-*x* is 1 and *y*-*y* is 4. In this case, ***a b c d*** can form a CG.

References in a CG have three important properties. First, the order of references must be exactly the same at each repetition (e.g. *d* must follow *c*, *c* follows *b* and *b* follows *a*). Second, references in a CG can interleave with other references (e.g. *x*, *y*). These references, however, are irregular and difficult to predict accurately, and will be excluded by the criteria of same reuse distance. Third, since we are interested in capturing references with long reuse distances for prefetching, the same reference (i.e. to the same block) usually does not appear twice in one CG.

To demonstrate the existence of CGs, we plot reuse distances of 3000 nearby references from three SPEC2000 applications, Mcf, Ammp and Parser in Figure 2-2. We randomly select 3000 memory references for each application, and compute the reuse distance of each reference. Note that references with short reuse distances (e.g. < 512), which are frequent due to temporal and spatial localities of memory references, can be captured by small caches and thus are filtered.

The existence of CGs is quite obvious from these snapshots. Mcf has a huge CG with a reuse distance of over 60,000. The reuse distance is so large that a reasonable sized L2 cache (<4MB, if each memory block is 64B) will not keep those blocks in it. So those accesses are likely to be L2 misses. Ammp shows four large CGs along with a few small ones. And Parser has many small CGs. Other applications also show the CG behavior. We only present three examples due to the space limit.

The next important question is whether there exist strong correlations among references in a CG, i.e. when a member of a captured CG is referenced, the other members will be referenced in the near future.

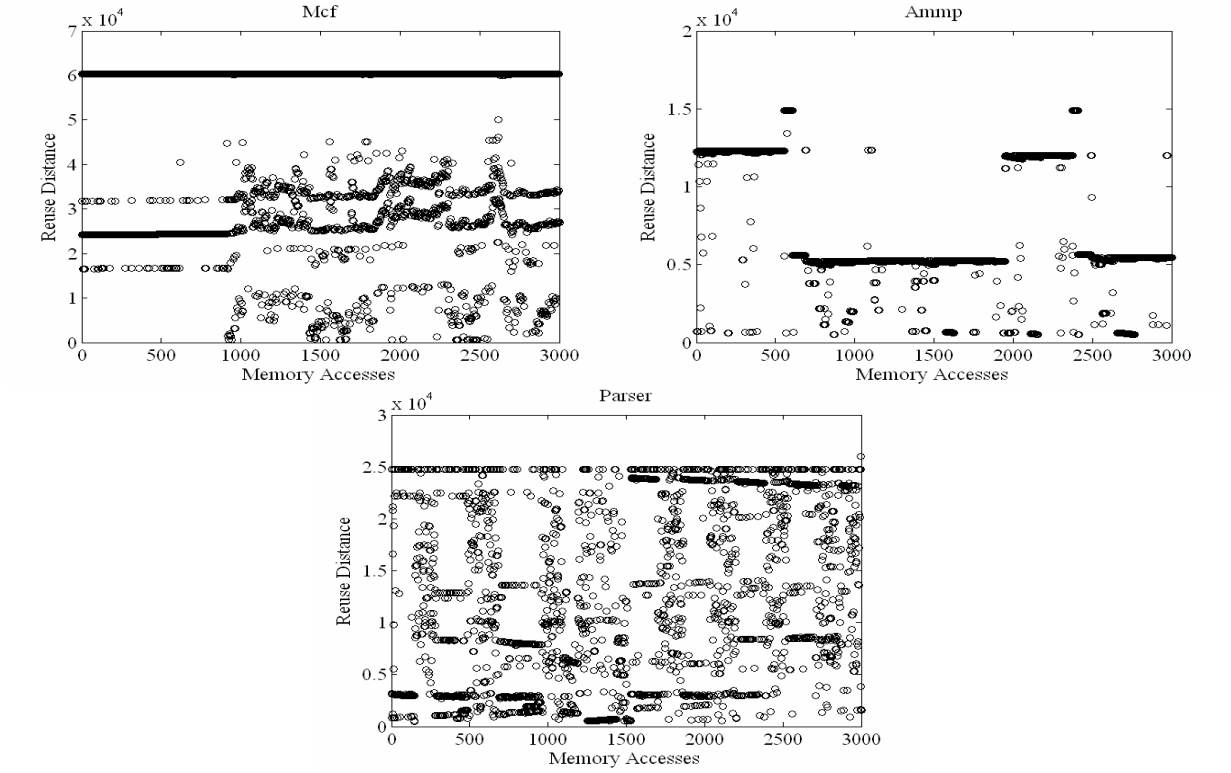


Figure 2-2. Reuse distances for Mcf, Ammp and Parser.

We can answer this question by measuring the *pair-wise correlation A->B between adjacent references* in a CG. (This is similar to the miss correlation in [37].) The accuracy of *B* being referenced immediately after the re-reference of *A* provides a good locality measurement. Considering that a CG {*a*, *b*, *c*, *d*} has been captured, and its reference order *a->b*, *b->c*, and *c->d* will be recorded in a history table. For an access that hits the table, we verify whether the actual next access is the next access that has been recorded. For example, if access *a* happens again, we need to verify whether *b* is the next access. If so, we count it as an accurate prediction based on the CG.

We can also relax the reuse distance requirement allowing a small variance in reuse distances. We define CG-N as CGs, in which nearby references with reuse distances that are within plus or minus N. A smaller N means more restricted CGs, potentially more accurate, while a larger N means more relaxed CGs, possibly including more members. CG-0 is the most

restricted one, representing the original same-distance CG, while CG-∞ is the most relaxed one that has a single CG including all nearby references with long block reuse distances.

Figure 2-3 shows the accuracy that the real next access of a member in a CG is indeed the next access being captured and saved for CG-0, CG-2, CG-8 and CG-∞ for all the individual applications. In this figure, there are 4 bars for each application representing the accuracy of CG-0, CG-2, CG-8 and CG-∞ from left to right. In general, CG-0, CG-2 and CG-8 exhibit strong repeated reference behaviors among members in a CG than CG-∞. Ammp and Art show nearly perfect correlations with about 98% of accuracy regardless of the reuse distance requirement. Those two applications are floating point applications with regular array accesses, which are easy to predict. All other applications also demonstrate strong correlations for CG-0, CG-2 and CG-8. As expected, CG-0 shows stronger correlations than other weaker forms of CGs, while CG-∞, which is essentially the same as the adjacent cache-miss correlation, shows very poor correlations. For instance, the accuracy of CG-0 for Mcf is about 78%, while the accuracy of CG-∞ is only 30%. The gap between CG-0 and CG-2/CG-8 is rather narrow for Mcf, Vortex, and Bzip2, suggesting a weaker form of CGs may be preferable for covering more references. A large gap is observed between CG-0 and other CGs in Twolf, Parser, and Gcc indicating CG-0 is more accurate for prefetching for those applications.

Based on these behaviors, we can safely conclude that members in a CG exhibit a highly repeated access pattern, i.e. *whenever a member in a CG is referenced, the remaining members will likely be referenced in the near future according to the previous accessing sequence*. We call such highly repeated patterns *coterminous locality*. Based on the existence of highly-repeated coterminous locality within members in CGs, we can design an accurate prefetching scheme to capture CGs, and prefetch members in CGs.

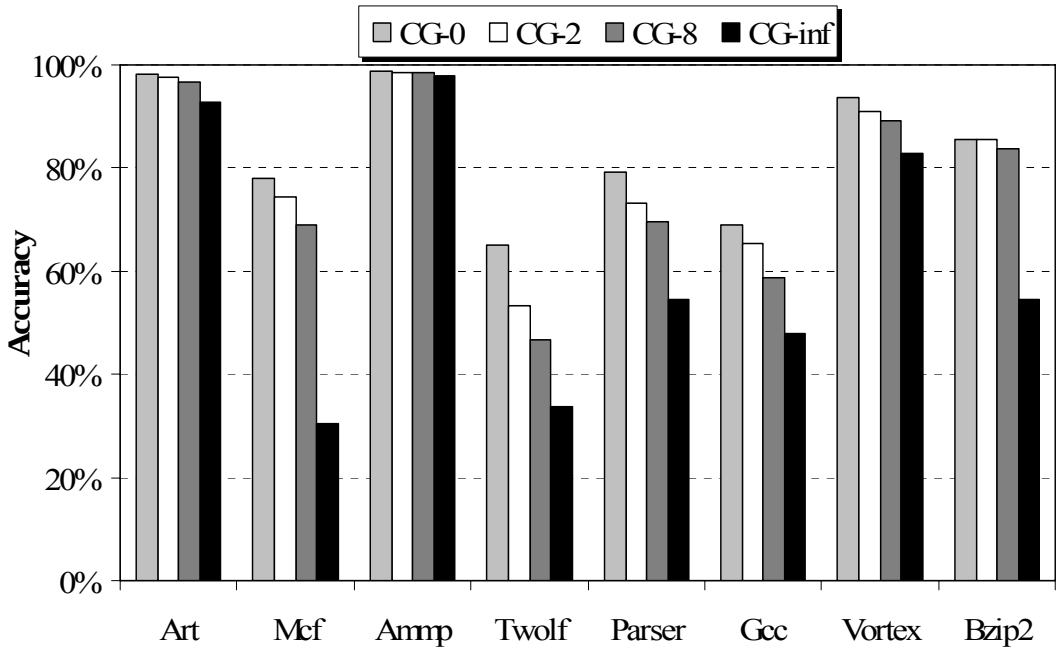


Figure 2-3. Strong correlation of adjacent references within CGs.

### 2.3 Memory-side CG-prefetcher on CMPs

Due to shared cache contentions on CMPs, it is more beneficial to prefetch those L2 misses to improve the overall performance. Our CG-prefetcher records L2 misses, captures CGs from the L2 miss sequence and prefetches members in CGs to reduce the number of expensive off-chip memory accesses. Since the memory controller sees every L2 miss directly, we integrate the CG-prefetcher into the memory controller. This is the memory-side CG-prefetcher.

A memory-side CG-prefetcher is attractive for several reasons [67]. First, it minimizes changes to the complex processor pipeline along with any associated performance and space overheads. Second, it may use the DRAM array to store necessary state information with minimum cost. A recent trend is to integrate the memory controller in the processor die to reduce interconnect latency. Nevertheless, such integration has minimal performance implication on implementing the CG-prefetcher in the memory controller. Note that although the CG-prefetcher

is suitable on uni-processor systems as well, it is more appealing on emerging CMPs with extra resource contentions and constraints due to its high accuracy and low overhead.

### 2.3.1 Basic Design of CG-Prefetcher

The structure of a CG-prefetcher is illustrated in Figure 2-4. There are several main functions: to capture nearby memory references with equal reuse distance, to form CGs, to efficiently save CGs in a history table for searching, and to update CGs and keep them fresh.

To capture nearby memory references with same distance, a *Request FIFO* records the block addresses and their reuse distances of recent main memory requests. A CG starts to form once the number of requests with the same reuse distance in the *Request FIFO* exceeds a certain threshold (e.g. 3), which controls the aggressiveness of forming a new CG. The size of the FIFO determines the adjacency of members, and usually it is small (e.g. 16). A flag is associated with each request indicating whether the request is matched. The matched requests in the FIFO are copied into a *CG Buffer* waiting for the CG to form. The size of the *CG buffer* determines the maximum number of members in a CG, which can control the timeliness of prefetches. A small number of *CG Buffers* can be implemented to allow multiple CGs to form concurrently. A CG is completed and will be saved when either the *CG Buffer* is full or a new *CG Buffer* is needed when a new CG is identified from the *Request FIFO*.

To efficiently save CGs, we introduce *Coterminous Group History Table (CGHT)*, a set-associative table indexed by block addresses, so that every member in a CG can be found very fast. A unidirectional pointer in each entry links the members in a CG. This link-based CGHT permits fast searching of a CG from any member in the group, thus allows to prefetch a CG starting from any member. When the CGHT becomes full, either the LRU entries are replaced and removed from the existing CGs, or the conflicting new CG members are dropped to avoid potential thrashing.

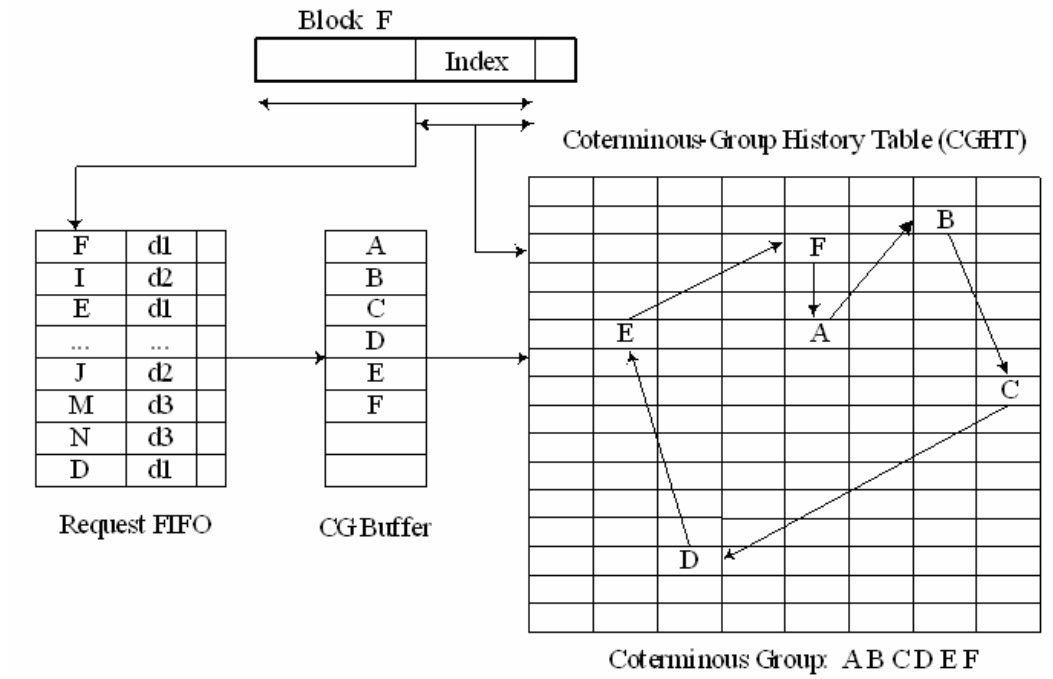


Figure 2-4. Diagram of the CG prefetcher.

Any existing CGs can change dynamically over a period of time. Updating CGs in the CGHT dynamically is difficult without precise information on when a member leaves or joins a group. Another option is to simply flush the CGHT periodically based on the number of executed instructions, memory references, or cache misses. However, a penalty will be paid to reestablish the CGs after each flush. Note that a block can appear in more than one CG in the CGHT without updating. This is because a block reference can leave a CG and become a member of a new CG, while the CGHT may still keep the old CG. On multiple hits, either the most recent CG or all the matched CGs may be prefetched.

Table 2-1 gives an example to illustrate the operations of CGs in Figure 2-4 step by step. We assume three accesses with the same reuse distance are requested to start forming a CG. We also assume memory accesses A, B and C have reuse distance *d1*, and have already been identified as a CG in the *Request FIFO*, and moved to the *CG Buffer*. So the initial state is that the *CG Buffer* records A, B and C, as well as the current reuse distance *d1*.

Table 2-1. Example operations of forming a CG

| Access | Reuse distance | Event | Comment |
|---|---|---|---|
| D | d1 | Put D in CG Buffer | d1 matches the reuse distance of current CG |
| N | d3 | None | d3 is not equal to d1 |
| M | d3 | None | Only two same reuse distance accesses, N and M |
| J | d2 | None | d2 is not equal to d1 |
| E | d1 | Put E in CG Buffer | d1 matches the reuse distance of current CG |
| I | d2 | None | Only two same reuse distance accesses, J and I |
| F | d1 | Put F in CG Buffer | d1 matches the reuse distance of current CG |

Table 2-1 simulates the situation when accesses D, N, M, J, E, I, F come one by one. D, E, and F have the same reuse distance *d1*, and will be recorded in the *CG Buffer* step by step. Although N and M have the same reuse distance *d3*, they cannot start to form a new CG until another access with reuse distance *d3* together with N and M appear in the Request FIFO at the same time. If this does happen, the current CG will be moved to the CGHT to make space for the newest CG. Once a L2 miss hits the CGHT, the entire CG can be identified and prefetched by following the circular links. In Figure 2-4, for instance, a miss to block F will trigger prefetches of A, B, C, D, and E in order.

### 2.3.2 Integrating CG-prefetcher on CMP Memory Systems

There are several issues to integrate the CG-prefetcher into the memory controller. The first key issue is to determine the block reuse distance without seeing all processor requests at the memory controller. A *global miss sequence number* is used. The memory controller assigns and saves a new sequence number to each missed memory block in the DRAM array. The reuse distance can be approximated as the difference of the new and the old sequence numbers.

For a 128-byte block with a 16-bit sequence number, a reuse distance of 64K blocks, or an 8MB working set can be covered. The memory overhead is merely 1.5%. When the same distance requirement is relaxed, one sequence number can be for a small number of adjacent requests, which will expand the working set coverage and/or reduce the space requirement.

Figure 2-5 shows the CG-prefetcher in memory system. To avoid regular cache-miss requests from different cores disrupting one another for establishing the CGs [70], we construct a private CG-prefetcher for each core.

Each CG-prefetcher has a *Prefetch Queue (PQ)* to buffer the prefetch requests (addresses) from the associated prefetcher. A shared *Miss Queue (MQ)* stores regular miss requests from all cores for accessing the DRAM channels. A shared *Miss Return Queue (MRQ)* and a shared *Prefetch Return Queue (PRQ)* buffers the data from the miss requests and the prefetch requests for accessing the memory bus.

We implement a private PQ to prevent prefetch requests of one core from blocking those from other cores. The PQs have lower priority than the MQ. Among the PQs, a round-robin fashion is used. Similarly, the PRQ has lower priority than the MRQ in arbitrating the system bus. Each CG-prefetcher maintains a separate sequence number for calculating the block reuse distance.

When a regular miss request arrives, all the PQs are searched. In case of a match, the request is removed from the PQ and is inserted into the MQ, gaining a higher priority to access the DRAM. In this case, there is no performance benefit since the prefetch of the requested block has not been initiated. If a matched prefetch request is in the middle of fetching the block from the DRAM, or is ready in the PRQ, waiting for the shared data bus, the request will be redirected to the MRQ for a higher priority to arbitrate the data bus. Variable delay cycles can be saved depending on the stage of the prefetch request. The miss request is inserted into the MQ normally when no match is found.

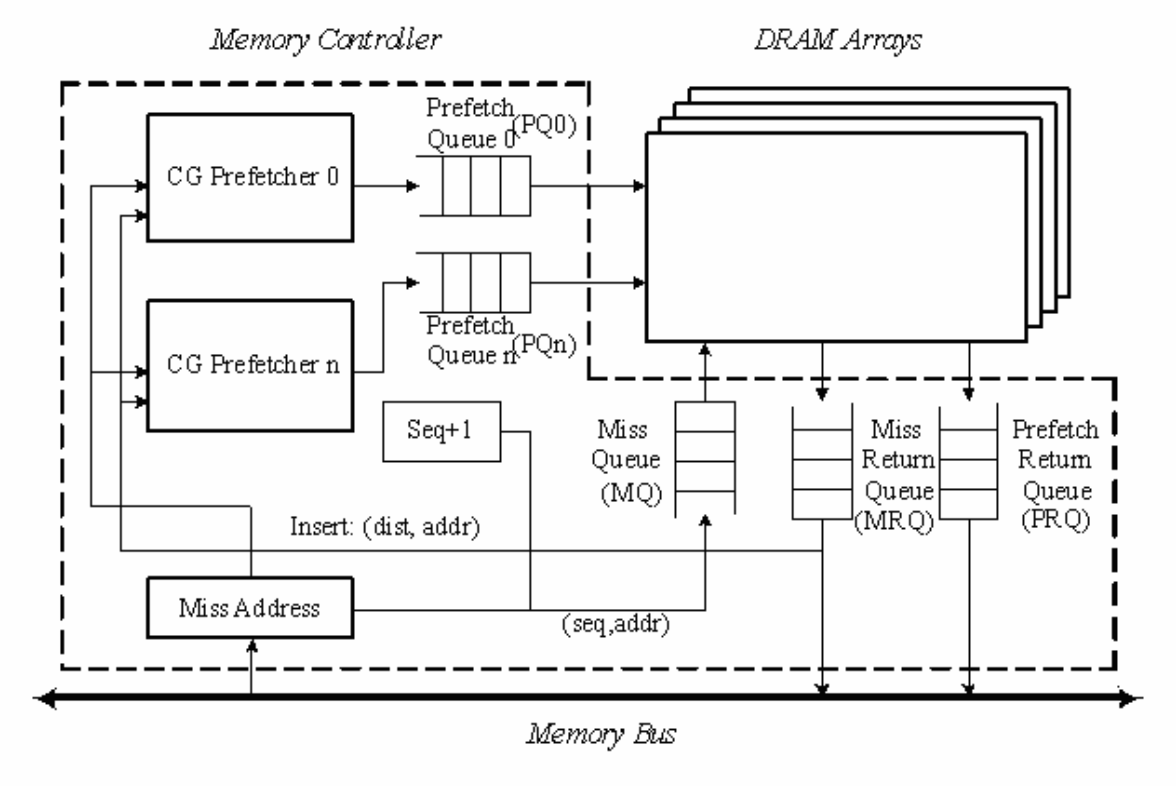


Figure 2-5. Integration of the CG-prefetcher into the memory controller.

A miss request can trigger a sequence of prefetches if it hits the CGHT. The prefetch requests are inserted into the corresponding PQ. If the PQ or the PRQ is full, or if a prefetch request has been initiated, the prefetch request is simply dropped. In order to filter the prefetched blocks already located in processor's cache, a topologically equivalent directory of the lowest level cache is maintained in the controller (not shown in Figure 2-5). The directory is updated based on misses, prefetches, and write-backs to keep it close to the cache directory. A prefetch is dropped in case of a match. Note that all other simulated prefetchers incorporate the directory too.

### 2.4 Evaluation Methodology

We simulate 2-core and 4-core CMPs with a 1MB, 8-way, shared L2 cache. Please note that the CG prefetcher can also be applied to other L2 organizations. We add an independent processor-side stride prefetcher to each core. All timing delays of misses and prefetches are

carefully simulated. Due to a slower clock of the memory controller, the memory-side prefetchers initiate one prefetch every 10 processor cycles. The queue size for MQ, PQi, PRQ and MRQ is all 16 enties. We use all 2-application, and 4-application workload mixes described in Chapter 1 for this work.

The performance results of the proposed CG-prefetcher are compared against a pair-wise miss-correlation prefetcher (MC-prefetcher), a prefetcher based on the last miss stream (LS-prefetcher), and a hot-stream prefetcher (HS-prefetcher). A processor-side stride prefetcher is included in all simulated memory-side prefetchers. Descriptions of these prefetchers are given next.

**Processor-side Stride prefetcher (Stride-prefetcher)**. The stride-prefetcher identifies and prefetches sequential or stride memory access patterns for specific PCs (program counters) [46]. It has 4k-entry PCs with each entry maintaining four previous references of that PC. Four successive prefetches are issued, whenever four stride distances of a specific PC are matched.

**Memory-side Miss-Correlation (MC) prefetcher**. The MC-prefetcher records pair-wise miss correlations A->B in a history table, and prefetches B if A happens again [37]. Each core has a MC-prefetcher with a 128k-entry 8 set-associative history table. Each miss address (each entry) records 2 successive misses. Upon a miss, the MC-prefetcher prefetches two levels in depth, resulting in a total of up to 6 prefetches.

**Memory-side Hot-Stream (HS) prefetcher**. The HS-prefetcher records a linear miss stream in a history table, and dynamically identifies and prefetcher repeated access sequences [23]. It is simulated based on a Global History Buffer [54], [78], with 128k-entry FIFO and 64k-entry 16 set-associative miss index table for each core. On every miss, the index and the FIFO are searched sequentially to find all recent streams that begin with the current miss. If the first 3

misses of any two streams match, the matched stream is prefetched. The length of each stream is 8.

**Memory-side Last-Stream (LS) prefetcher**. The LS-prefetcher is a special case of the HS-prefetcher [23], where the last miss stream is prefetched without any further qualification. It has the same implementation as the HS-prefetcher.

**Memory-side Coterminous Group (CG) prefetcher**. We use CG-2 to get both high accuracy and decent coverage of misses. The CGHT is 16k entries per core, with 30 bits (16-way set-associative) per entry. We use a 16-entry Request FIFO and four 8-entry CG-Buffers. A CG can be formed once three memory requests in the Request FIFO satisfy the reuse distance requirement. Each CG contains up to 8 members. The CGHT is flushed periodically every 2 million misses from the corresponding core.

Table 2-2 summaries the extra space overhead to implement various memory-side prefetchers. Note that the space overhead for processor-side stride prefetcher is minimal, thus has not been included.

## 2.5 Performance Results

In Figure 2-6 (A) and Figure 2-6 (B), the combined IPC of Stride-, MC-, HS-, LS-, and CG-prefetchers, normalized to that of the baseline model without any prefetching, are presented for 4-core CMPs and 2-core CMPs respectively. Please note in all the following figures, we simplify Stride-prefetcher to Stride, MC-prefetcher to MC, HS-prefetcher to HS, LS-prefetcher to LS and CG-prefetcher to CG. We include the normalized combined IPC of the base line model without any prefetching for comparison purpose (with the total height of 1). Note that the absolute combined IPCs for the baseline model were given in Figure 2-1. Also, a separate processor-side stride prefetcher is always running with MC-, HS-, LS-, and CG-prefetcher to prefetch blocks with regular access patterns.

Table 2-2. Space overhead for various memory-side prefetcher

| Prefetcher | Memory controller (SRAM) per core | DRAM |
|---|---|---|
| CG-prefetcher | 60KB (16K*30bit/8) | 3% |
| MC-prefetcher | 2MB (128K*2*64bit/8) | 0 |
| HS-prefetcher | 920KB(128K*43bit/8+64K*29bit/8) | 0 |
| LS-prefetcher | 920KB(128K*43bit/8+64K*29bit/8) | 0 |

Each bar is broken into the contributions made by individual workloads in the mix. The total height represents the overall normalized IPC or IPC speedup. For example, a total height of 1.2 means a 20% improvement in IPS as compared with the base IPC without any prefetching, given in Figure 2-1. Please note a normalized IPC of less than one means the prefetcher degrades the overall performance.

Several observations can be made. First, most workload mixes show performance improvement for all five prefetching techniques. In general, the CG-prefetcher has the highest overall improvement, followed by the LS-, the HS-, and the MC-prefetchers. Two workload mixes Art/Twolf and Mcf/Twolf show a performance loss for most prefetchers. Our studies indicate that Twolf has irregular patterns, and hardly benefits from any of the prefetching schemes. Although Art and Mcf are well performed, the higher IPC of Twolf dominates the overall IPC speedup.

Second, the CG-prefetcher is a big winner for the MEM workloads with speedup of 40% in average, followed by the LS-prefetcher with 30%, the HS-prefetcher with 24% and the MC-prefetcher with 18%. The MEM workloads exhibit heavier cache contentions and misses. Therefore, the accurate CG-prefetcher benefits the most for this category.

Third, the CG-prefetcher generally performs better in the MIX and the CPU categories. However, the LS-prefetcher slightly outperforms the CG-prefetcher in a few cases. With lighter memory demands in these workload mixes, the LS-prefetcher can deliver more prefetched blocks with a smaller impact on cache pollutions and memory traffic.

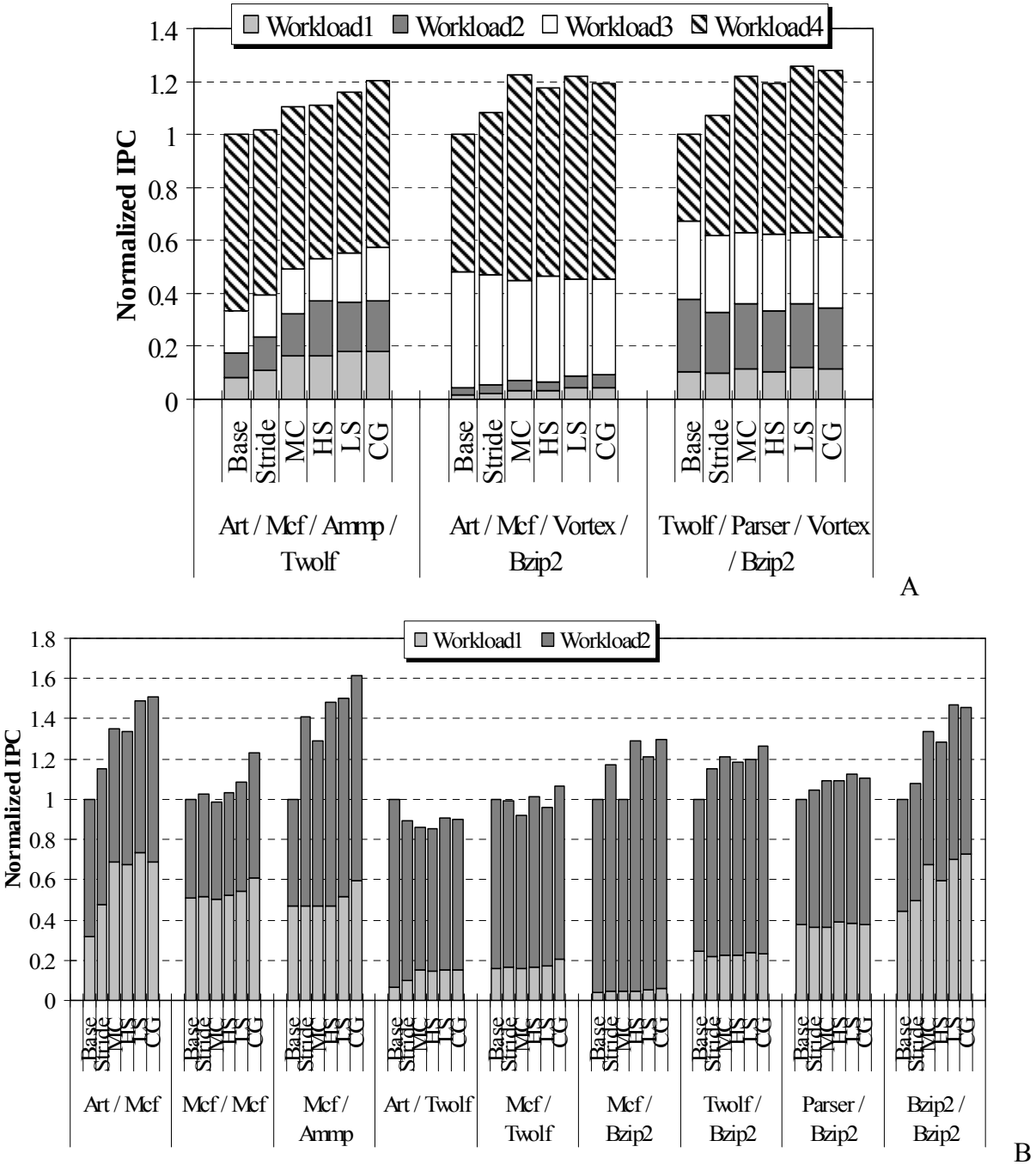


Figure 2-6. Normalized combined IPCs of various prefetchers. (Normalized to baseline). A) 4-workload mix running on 4-core CMPs. B) 2-workload mix running on 2-core CMPs.

It is important to note that the normalized IPC speedup creates an unfair view when comparing mixed workloads on multi-cores. For example, in Art/Mcf/Vortex/Bzip2, the normalized IPCs of individual workloads are measured at 3.16 for Art, 1.41 for Mcf, 0.82 for

Vortex, and 1.42 for Bzip2 with the CG-prefetcher, and 2.39 for Art, 1.22 for Mcf, 0.86 for Vortex, and 1.49 for Bzip2 with the MC-prefetcher. Therefore, the average individual speedups of the four workloads, according to equation (2-1), are 1.70 for the CG-prefetcher and 1.49 for the MC-prefetchers. However, their normalized IPCs are only 1.20 and 1.22. Given the fact that Vortex and Bzip2 have considerably higher IPC than those of Art and Mcf, the overall IPC improvement is dominated by the last two workloads. This is true for other workload mixes.

In Figure 2-7, the average speedup of two MEM and two MIX workload mixes are shown according to the equation (2-1). Please recall that all the applications in a MEM workload mix are memory-intensive, and a MIX workload mix contains both memory-intensive and cpu-intensive applications.

$$Average\ Speedup = \sum_{i=0}^{n} \frac{IPC_i\ with\ prefetch}{IPC_i\ without\ prefetch} \tag{2-1}$$

where n is the number of workloads.

Comparing with the measured IPC speedups in Figure 2-6, significantly higher average speedups are achieved by all prefetchers. For Art/Twolf, the average IPC speedups are 48% for the MC-prefetcher, 44% for the HS-prefetcher, 52% for the LS-prefetcher and 51% for the CG-prefetcher, instead of the IPC degradations as shown in Figure 2-6.

Figure 2-8 shows the accuracy and coverage of different prefetchers. In this figure, each bar is broken down into 5 categories from bottom to top for each prefetcher: misses, partial hits, miss reductions, extra prefetches, and wasted prefetches. The descriptions of each category are listed as follows.

- The misses are those main memory accesses that have not been covered by the prefetchers.

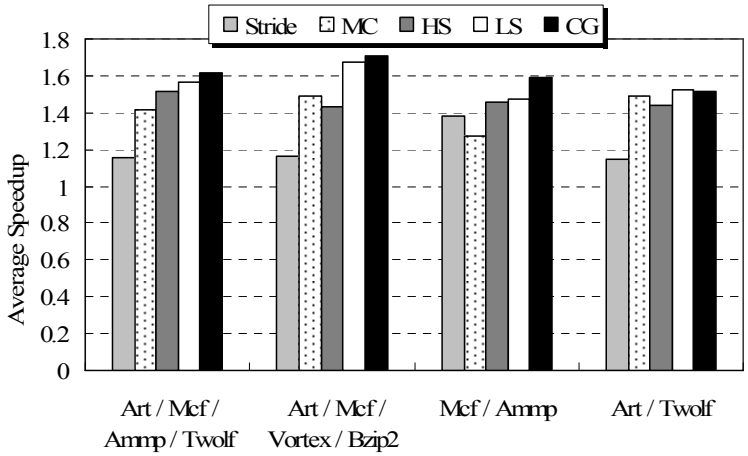


Figure 2-7. Average speedup of 4 workload mixes.

- The partial hits refer to memory accesses with a part of off-chip access latency being saved by earlier but incomplete prefetches. The earlier prefetches have been issued to the memory, but the blocks have not arrived in the L2 cache.

- The miss reductions are those that have been fully covered by prefetches. Those accesses are successfully and completely changed from L2 misses to L2 hits.

- The extra prefetches represent the prefetched blocks that are replaced before any usage. Those useless prefetched blocks will pollute the limited cache space and waste the limited bandwidth.

- The wasted prefetches refer to the prefetched blocks that are presented in L2 cache already when those blocks arrive in the L2 cache because of the mis-prediction of the shadow directory at the memory side, which wastes memory bandwidth.

The sum of the misses, partial hits, and miss reductions is equal to the number of misses of the baseline without prefetching, which is normalized to 1 in the figure. And the sum of extra prefetches and wasted prefetches, normalized to baseline misses, is the extra memory traffic.

According to the above definition, the accuracy of a prefetcher can be described as Equation (2-2):

$$Coverage = \frac{Misses\ reductions + Partial\ hits}{Misses\ reductions + Partial\ hits + Misses} \tag{2-2}$$

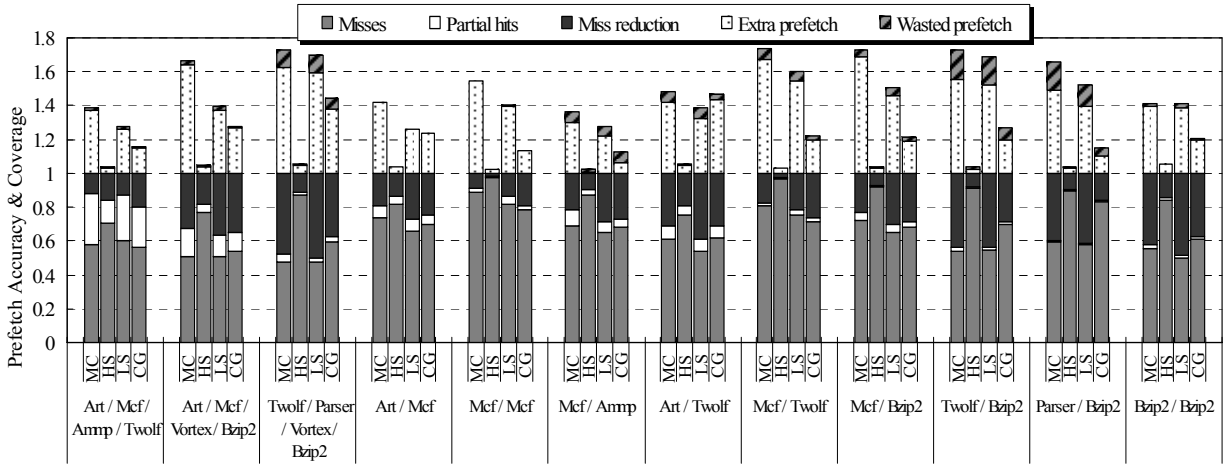


Figure 2-8. Prefetch accuracy and coverage of simulated prefetchers.

The coverage of a prefetcher can be calculated as Equation (2-3):

$$Accuracy = \frac{Extra\ \ prefetch + Wasted\ \ prefetch}{Misses\ reductions + Partial\ hits + Extra\ \ prefetch + Wasted\ \ prefetch} \quad (2\text{-}3)$$

Overall, all prefetchers show a significant coverage, reduction of cache misses, ranging from a few percent to as high as 50%. The MC-, LS- and CG-prefetchers have better coverage than HS-prefetcher, since HS-prefetcher only identifies exactly repeated accesses. On the other hand, in contrast to the MC- and the LS-prefetcher, the HS- and the CG-prefetcher carefully qualify members in a group that show highly repeated patterns for prefetching. The MC- and the LS-prefetcher generate significantly higher memory traffic than the HS- and the CG-prefetcher. On the average, the HS-prefetcher has the least extra traffic of about 4%, followed by 21% for the CG-prefetcher, 35% for the LS-prefetcher, and 52% for the MC-prefetcher. The excessive memory traffic by the LS- and the MC-prefetcher does not turn proportionally into a positive reduction of the cache miss. In some cases, the impact is negative mainly due to the cache pollution problem on CMPs. Between the two, the LS-prefetcher is more effective than the MC-prefetcher indicating prefetching multiple successor misses may not be a good idea. The HS-

prefetcher has the highest accuracy. However, the low miss coverage limits its overall IPC improvement.

The reuse distance constraint of forming a CG is simulated and the performance results of CG-0, CG-2 and CG-8 are plotted in Figure 2-9. With respect to the normalized IPCs, the results are mixed, shown in Figure 2-9 (A). Figure 2-9 (B) further plots the coverage and accuracy of different CGs.

It is evident that the reuse distance constraint is the tradeoff between accuracy and coverage. In generally, larger distance means higher coverage, but lower accuracy. CG-2 seems to be better than CG-8 for workload mixes having higher L2 misses, such as Mcf/Ammp and Mcf/Mcf. We selected CG-2 to represent the CG-prefetcher due to its slightly better IPCs than that of CG-0 and considerably less traffic than that of CG-8. Note that we omit CG-4, which has similar IPC speedup in comparison with CG-2, but generates more memory traffic.

The impact of group size is evaluated as shown in Figure 2-10. Two workload mixes in the MEM category, Art/Mcf/Ammp/Twolf and Mcf/Ammp, and two in the MIX category, Art/Mcf/Vortex/Bzip2 and Art/Twolf are chosen due to their high memory demand. The measured IPCs decrease slightly or remain unchanged for the two 4-workload mixes, while they increase slightly with the two 2-workload mixes. Due to cache contentions, larger groups generate more useless prefetches. The group size of 8 shows a balance of high IPCs with low overall memory traffic.

Figure 2-11 plots the average speedup of CG with respect to Stride-only for different L2 cache sizes from 512KB to 4MB. As observed, the four workload mixes behave very differently with respect to different L2 sizes.

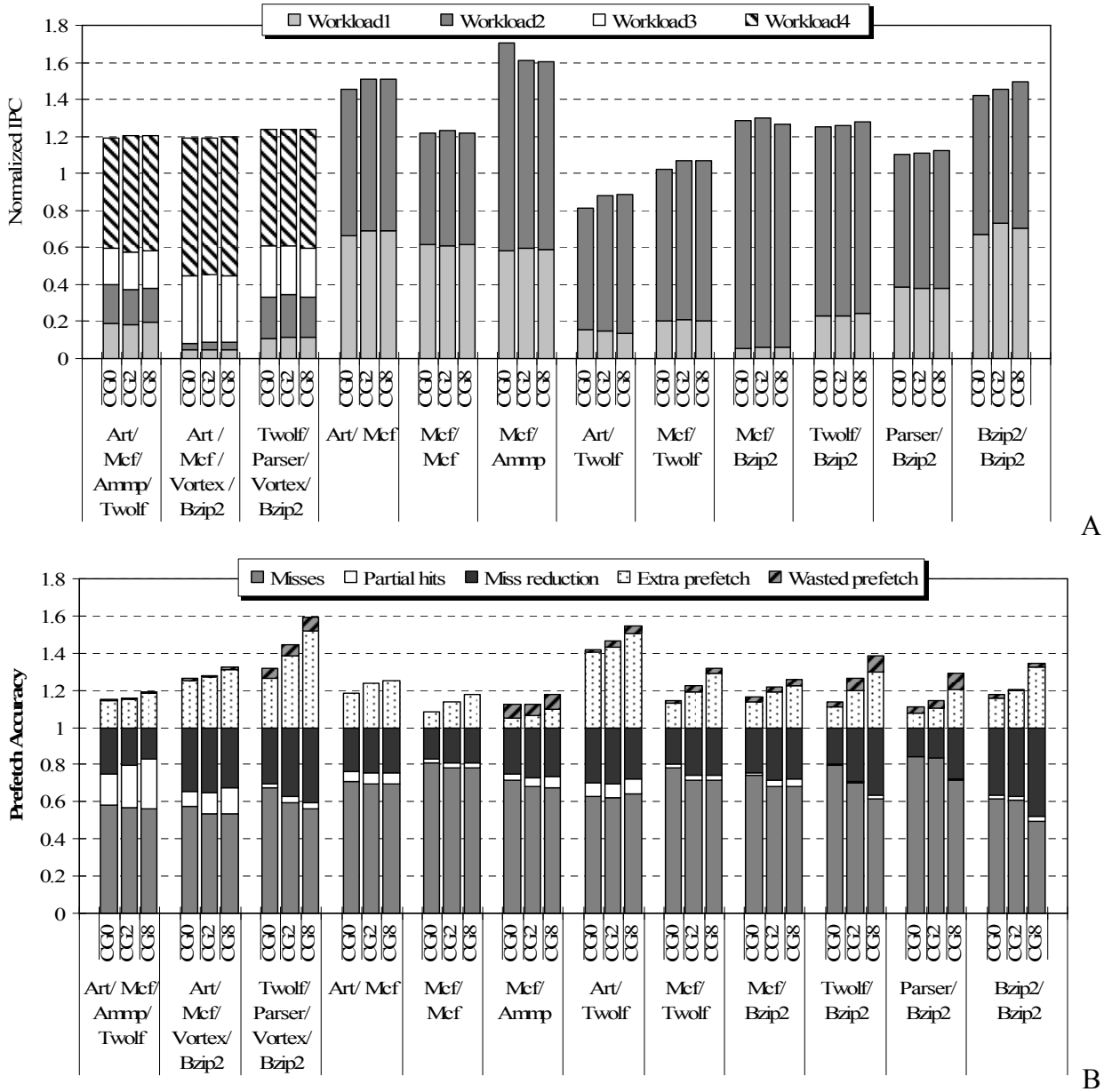


Figure 2-9. Effect of distance constrains on the CG-prefetcher. A) Normalized IPC. B) Accuracy and traffic.

For Art/Mcf/Vortex/Bzip2 and Art/Twolf, the average IPC speedups are peak at 1MB and 2MB respectively, and then drop sharply afterwards because of a sharp reduction of cache misses with larger caches. However, for the memory-bound workload mixes, Art/Mcf/Ammp/Twolf and Mcf/Ammp, the average speedups of median-size L2 are slightly less than those of smaller and larger L2.

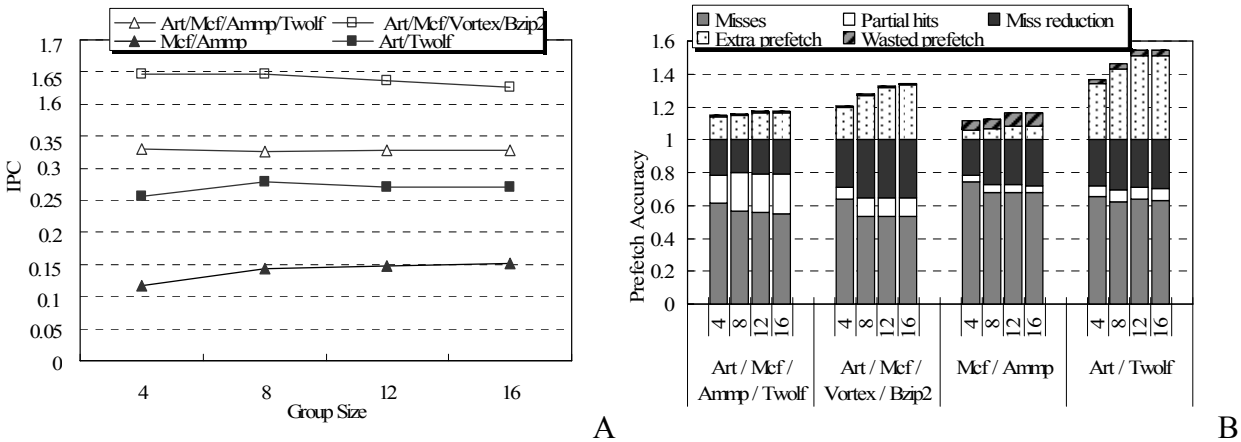


Figure 2-10. Effect of group size on the CG-prefetcher. A) Measured IPCs. B) Accuracy and traffic.

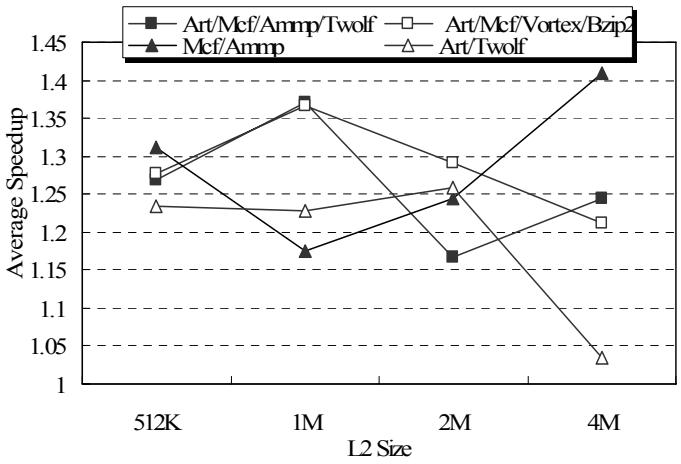


Figure 2-11. Effect of L2 size on the CG-prefetcher.

With smaller caches, the cache contention problem is so severe that a small percentage of successful prefetches can lead to significant IPC speedups. For median size caches, the impact of delaying normal miss due to conflicts with prefetches begins to compensate the benefit of prefetching. When the L2 size continues to increase, the number of misses decreases and it diminishes the effect of accessing conflicts. As a result, the average speedup increases again.

Given a higher demand for accessing the DRAM for the prefetching methods, we perform a sensitivity study on the DRAM channels as shown in Figure 2-12. The results indicate that the number of DRAM channels does show impacts on the IPCs and more so to the memory-bound

workload mixes. All four workload mixes perform poorly with 2 channels. However, the improvements are saturated about 4 to 8 channels.

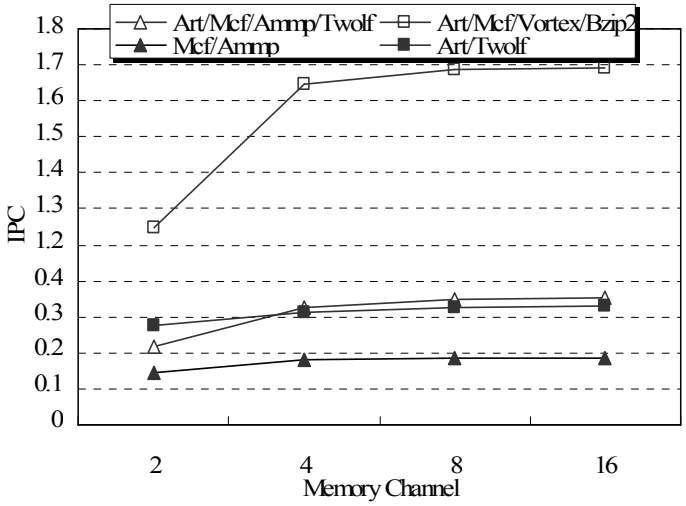


Figure 2-12. Effect of memory channels on the CG-prefetcher.

## 2.6 Summary

We have introduced an accurate CG-based data prefetching scheme on Chip Multiprocessors (CMPs). We have showed the existence of coterminous groups (CGs) and a third kind of locality, coterminous locality. In particular, the order of nearby references in a CG follows exactly the same order that these references appeared last time, even though they may be irregular. The proposed prefetcher uses CG history to trigger prefetches when a member in a group is re-referenced. It overcomes challenges of the existing correlation-based or stream-based prefetchers, including low prefetch accuracy, lack of timeliness, and large history. The accurate CG-prefetcher is especially appealing for CMPs, where cache contentions and memory access demands are escalated. Evaluations based on various SPEC CPU 2000 workload mixes have demonstrated significant advantages of the CG-prefetcher over other existing prefetching schemes on CMPs.

# CHAPTER 3
# PERFORMANCE PROJECTION OF ON-CHIP STORAGE OPTIMIZATION

Organizing on-chip storage space on CMPs has become an important research topic. Balancing between data accessibility due to wiring delay and the effective on-chip storage capacity due to data replication has been studied extensively. These studies must examine a wide-spectrum of the design space to have a comprehensive view. The simulation time is prohibitively long for these timing simulations, and would increase drastically as the number of cores increases. A great challenge is then how to provide an efficient methodology to study design choices of optimizing CMP on-chip storage accurately and completely, when the number of cores increases.

In the second work, we first develop an analytical model to assess general performance behavior with respect to data replications in CMP caches. The model injects replicas (replicated data blocks) into a generic cache. Based on the block reuse-distance histogram obtained from a real application, a precise mathematical formula is derived to evaluate the impact of the replicas. The results demonstrate that whether data replication helps or hurts L2 cache performance is a function of the total L2 size and the working set of the application.

To overcome the limitations of modeling, we further develop a single-pass stack simulation technique to handle shared and private cache organizations with the invalidation-based coherence protocol. The stack algorithm can handle complex interactions among multiple private caches. This single-pass stack technique can provide local/remote hit ratios and the effective cache size for a range of physical cache capacities. We also demonstrate that we can use the basic multiprocessor stack simulation results to estimate the performance of other interesting CMP cache organizations, such as shared caches with replication and private caches without replication.

We verify the results of the analytical data replication model and the single-pass global stack simulation with detailed execution-driven simulations. We show that the single-pass stack simulation produces small error margins of 2-9% for all simulated cache organizations. The total simulation times for the single-pass stack simulation and the individual execution-driven simulations are compared. For a limited set of the four studied cache organizations, the stack simulation takes about 8% of the execution-driven simulation time.

### 3.1 Modeling Data Replication

We first develop an abstract model independent of private/shared organizations to evaluate the tradeoff between the access time and the miss rate of CMP caches with respect to data replication. The purpose is to provide a uniform understanding on this central issue of caching in CMP that is present in most major cache organizations. This study also highlights the importance of examining a wide enough range of system parameters in the performance evaluation of any cache organization, which can be costly.

In Figure 3-1, a generic histogram of block reuse distances is plotted, where the reuse distance is measured by the number of distinct blocks between two adjacent accesses to the same block. A distance of zero indicates a request to the same block as the previous request. The histogram is denoted by *f(x)*, which represents the number of block references with reuse distance *x*.

For a cache size *S*, the total cache hits can be measured by $\int_0^S f(x)\,dx$, which is equal to the area under the range of the histogram curve from *0* to *S*. This well-known, stack distance histogram can provide hits/misses of all cache sizes with a fully-associative organization and the LRU replacement policy.

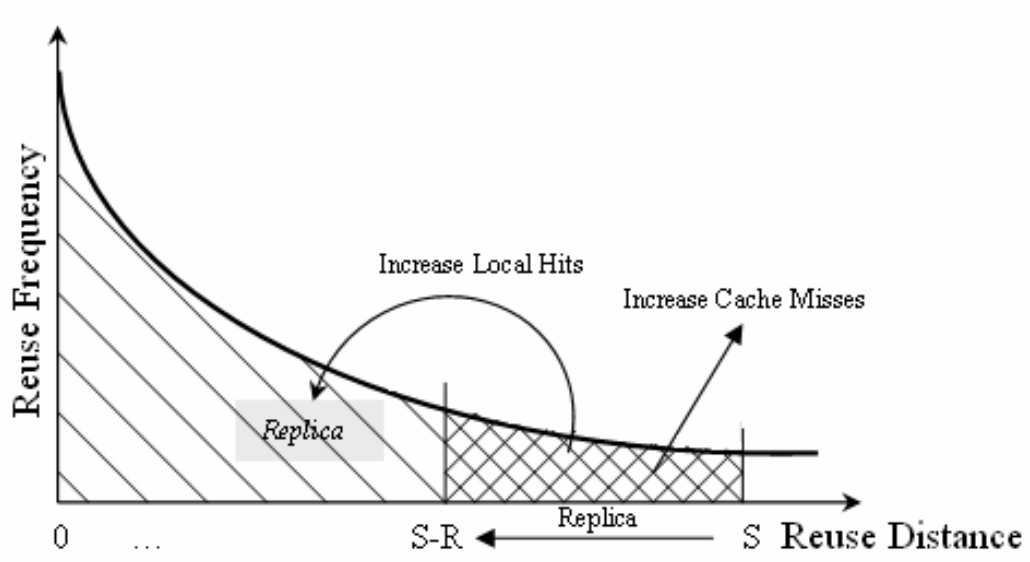

Figure 3-1. Cache performance impact when introducing replicas.

To model the performance impact of data replication, we inject replicas into the cache. Note that regardless the cache organization, replicas help to improve the local hit rate since replicas are created and moved close to the requesting cores. On the other hand, having replicas reduces the effective capacity of the cache, and hence, increases cache misses. We need to compare effect from the increase of local hits against that from the increase of cache misses.

Suppose we take a snapshot of the L2 cache and find a total of *R* replicas. As a result, only *S*-*R* cache blocks are distinct, effectively reducing the capacity of the cache. Note that the model does not make reference to any specific cache organization and management. For instance, it does not say where the replicas are stored, which may depend on factors such as shared or private organization. We will compare this scenario with the baseline case where all *S* blocks are distinct. First, the cache misses are increased by $\int_{S-R}^{S} f(x)\,dx$, since the total number of hits is now $\int_{0}^{S-R} f(x)\,dx$. On the other hand, the replicas help to improve the local hits. Among the $\int_{0}^{S-R} f(x)\,dx$ hits, a fraction *R*/*S* hits are targeting to the replicas. Depending on the specific cache organization, not all accesses to the replicas result in local hits. A requesting core may find a

replica in the local cache of another remote core, resulting in a remote hit. We assume that a fraction *L* accesses to replicas are actually local hits. Therefore, compared with the baseline case, the total change of memory cycles due to the creation of *R* replicas can be calculated by:

$$P_m \times \int_{S-R}^{S} f(x)dx \; - \; G_l \times \frac{R}{S} \times L \times \int_{0}^{S-R} f(x)dx \tag{3-1}$$

where *Pm* is the penalty cycles of a cache miss; and *Gl* is the cycle gain from a local hit. With the total number of memory accesses, $\int_0^\infty f(x)dx$, the average change of memory access cycles is equal to:

$$\left( P_m \times \int_{S-R}^{S} f(x)dx \; - \; G_l \times \frac{R}{S} \times L \times \int_{0}^{S-R} f(x)dx \right) \Big/ \int_0^\infty f(x)dx \tag{3-2}$$

Now the key is to obtain the reuse distance histogram *f(x)*. We conduct experiment using an OLTP workload [57] and collect its reuse distance histogram. With the curve-fitting tool of Matlab [52], we obtain the equation $f(x) = Aexp(-Bx)$, where $A = 6.084*10^6$ and $B = 2.658*10^{-3}$. This is shown in Figure 3-2, where the cross marks represent the actual reuse frequencies from OLTP and the solid line is the fitted curve. We can now substitute *f(x)* into equation (3-2) to obtain the average change in memory cycles as:

$$P_m \times \left( e^{-B(S-R)} - e^{-BS} \right) - G_l \times \frac{R}{S} \times L \times \left( 1 - e^{-B(S-R)} \right) \tag{3-3}$$

Equation (3-3) provides the change in L2 access time as a function of the cache area being occupied by the replicas. In Figure 3-3, we plot the change of the memory access time for three cache sizes, 2, 4, and 8 MB, as we vary the replicas' occupancy from none to the entire cache. In this figure, we assume *Gl*=15, *Pm*= 400, and we vary *L* with 0.25, 0.5 and 0.75 for each cache size. Note that negative values mean performance gain. We can observe that the performance of allocating L2 space for replicas for the OLTP workload varies with different cache sizes.

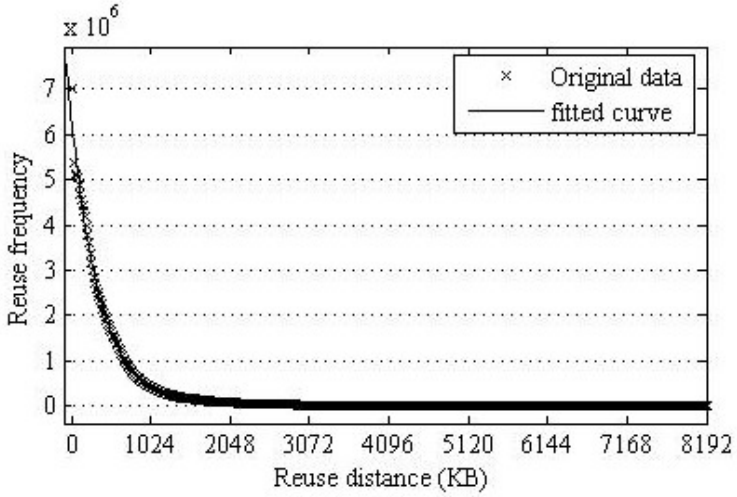


Figure 3-2. Curve fitting of reuse distance histogram for the OLTP workload.

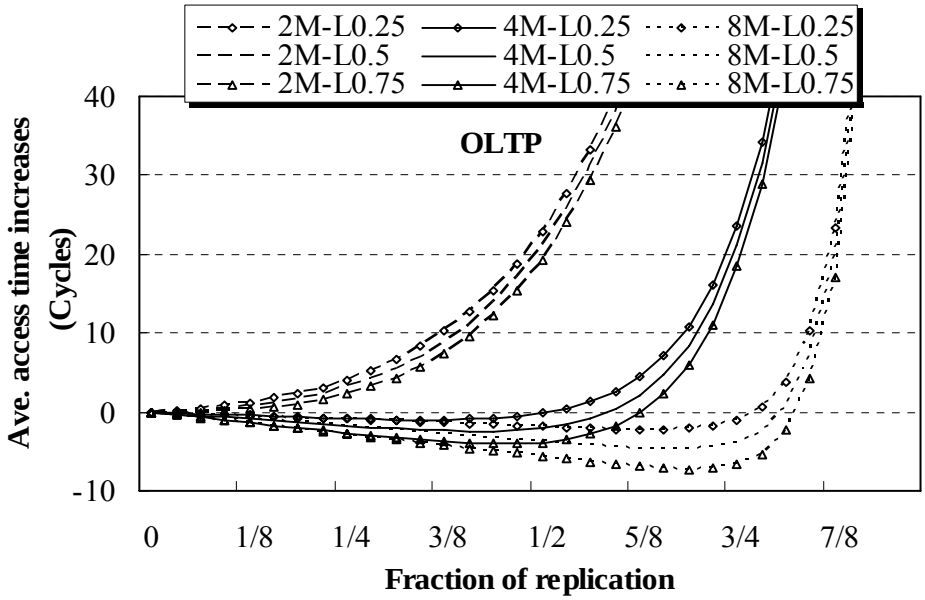


Figure 3-3. Performance with replicas for different cache sizes derived by the analytical model.

For instance, when $L = 0.5$, the results indicate no replication provides the shortest average memory access time for a 2MB L2 cache, while for larger 4MB and 8MB L2 caches, allocating 40% and 68% of the cache for the replicas has the smallest access time. These results are consistent with the reuse histogram curve shown in Figure 3-2. The reuse count approaches zero when the reuse distance is equal to or greater than 2MB. It increases significantly when the reuse distance is shorter than 2MB. Therefore, it is not wise to allocate space for the replicas when the

cache size is 2MB or less. Increasing *L* favors data replication slightly. For instance, for a 4MB cache, allocating 34%, 40%, 44% of the cache for the replicas achieves the best performance improvement of about 1, 3, and 5 cycles on the average memory access time for $L = 0.25$, 0.5 and 0.75 respectively. The performance improvement with data replication would be more significant when *Gl* increases.

The general behavior due to data replication is consistent with the detailed simulation result as will be given in Section 3.4. Note that the fraction of replicas cannot reach 100% unless the entire cache is occupied by a single block. Therefore, in Figure 3-3, the average memory time increase is not meaningful when the fraction of replicas is approaching to the cache size.

We also run the same experiment for two other workloads, Apache and SPECjbb. Figure 3-4 plots the optimal fractions of replication for all three workloads with cache size from 2 to 8MB and *L* from 0.25 to 0.75. The same behavior can be observed for both Apache and SPECjbb. Lager caches favor more replication. For example, with $L = 0.5$, allocating 13%, 50%, 72% space for replicas has the best performance for Apache, and 28%, 59%, 78% for SPECjbb. Also, increasing *L* favors more replication. With smaller working set, SPECjbb benefits replication the most among the three workloads.

It is essential to study a set of representative workloads with a spectrum of cache sizes to understand the tradeoff of accessibility vs. capacity on CMP caches. A fixed replication policy may not work well for a wide-variety of workloads on different CMP caches. Although mathematical modeling can provide understanding of the general performance trend, its inability to model sufficiently detailed interactions among multiple cores makes it less useful for accurate performance prediction. To remedy this problem, we will describe a global stack based simulation for studying CMP caches next.

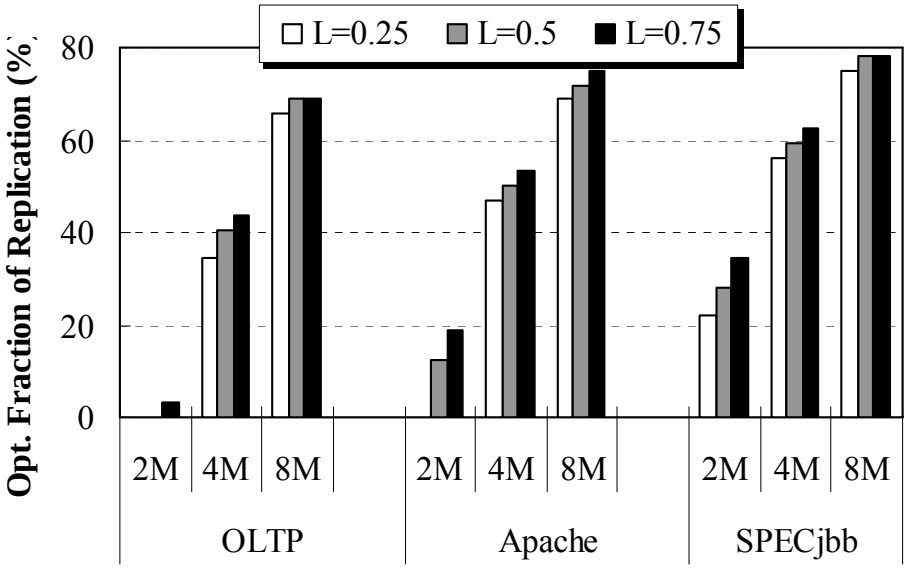


Figure 3-4. Optimal fraction of replication derived by the analytical model.

### 3.2 Organization of Global Stack

Figure 3-5 sketches the organization of the global stack, which records the memory reference history from all the cores.

In the CMP context, a block address and its core-id uniquely identify a reference, where the core-id indicates from which core the request is issued. Several independent linked lists are established in the global stack for simulating a *shared* and several per-core *private* stacks. Each stack entry appears exactly in one of the private stacks determined by the core-id, and may or may not reside in the shared stack depending on the recency of the reference. In addition, an address-based *hash* list is also established in the global stack for fast searches.

Since only a set of discrete cache sizes are of interest for cache studies, both the shared and the private stacks are organized as groups [43]. Each group consists of multiple entries for fast search during the stack simulation and for easy calculations of cache hits under various interesting cache sizes after the simulation.

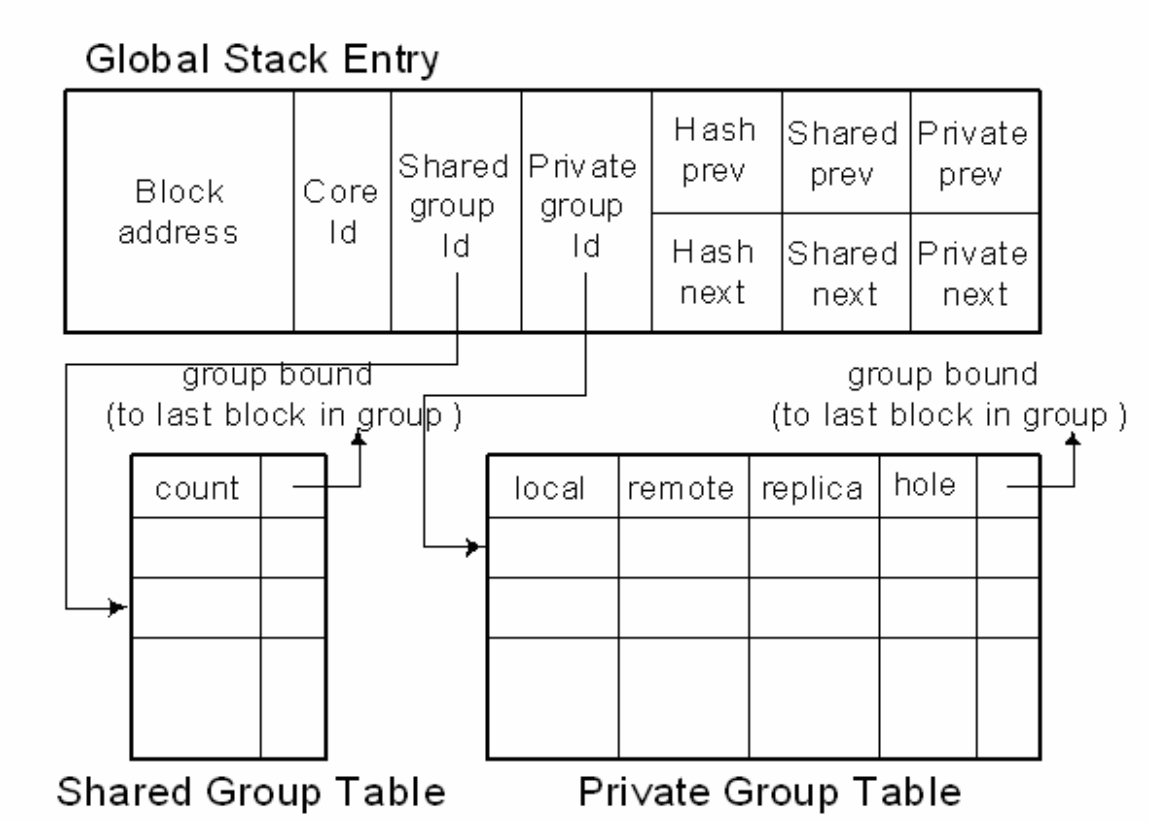


Figure 3-5. Single-pass global stack organization.

For example, assuming the cache sizes of interest are 16KB, 32KB, and 64KB. The groups can then be organized according to the stack sequence starting from the MRU entry with 256, 256, 512 entries for the first three groups, respectively, assuming the block size is 64B. Based on the stack inclusion property, the hits to a particular cache size are equal to the sum of the hits to all the groups accumulated up to that cache size. Each group maintains a reuse counter, denoted by *G1*, *G2*, and *G3*. After the simulation, the cache hits for the three cache sizes can be computed as *G1*, *G1+G2*, and *G1+G2+G3* respectively.

Separate shared and private group tables are maintained to record the reuse frequency count and other information for each group in the shared and private caches. A shared and a private group-id are kept in each global stack entry as a pointer to the corresponding group information in the shared and the private group table. The group bound in each entry of the group

table links to the last block of the respective group in the global stack. These group bounds provide fast links for adjusting entries between adjacent groups. The associated counters are accumulated on each memory request, and will be used to deduce cache hit/miss ratios for various cache sizes after the simulation. The following subsections provide detailed stack operations.

### 3.2.1 Shared Caches

Each memory block can be recorded multiple times in the global stack, one from each core according to the order of the requests. Intuitively, only the first-appearance of a block in the global stack should be in the shared list since there is no replication in a shared cache. A first-appearance block is the one that is most recently used in the global stack among all blocks with the same address.

The shared stack is formed by linking all the first-appearance blocks from MRU to LRU. Figure 3-6 illustrates an example of a memory request sequence and the operations to the shared stack. Each memory request isdenoted as a block address, A, B, C, …, etc., followed by a core-id. The detailed stack operations when B1 is requested are described as follows.

- Address B is searched by the hash list of the shared stack. B2 is found with the matching address. In this case, the reuse counter for the shared group where B2 resides, group 3, is incremented.
- B2 is removed from the shared list, and B1 is inserted at the top of the shared list.
- The shared group-id for B1 is set to 1. Meanwhile, the block located on the boundary of the first group, E1, is pushed to the second group. The boundary adjustment continues to the group where B2 was previously located.
- If a requested block cannot be located through the hash list, (i.e. the very first access of the address among any cores), the stack is updated as above without incrementing any reuse counters.
- After the simulation, the total number of cache hits for a shared cache that include exactly the first *m* groups is the sum of all shared reuse counters from group 1 to group *m*.

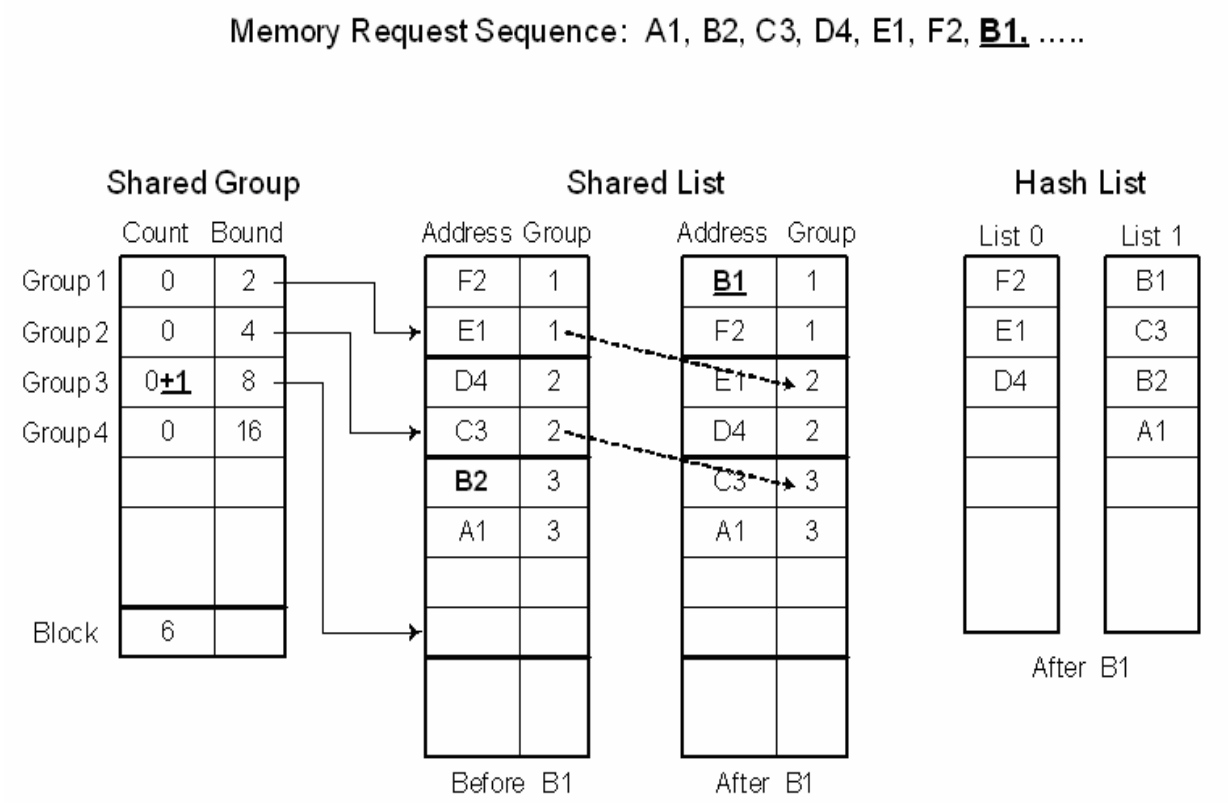


Figure 3-6. Example operations of the global stack for shared caches.

### 3.2.2 Private Caches

The construction and update of the private lists are essentially the same as those of the shared list, except that we link accesses from the same core together. We collect crucial information such as the local hits, remote hits, and number of replicas, with the help of the local, remote, and replica counters in the private group table. For simplicity, we assume these counters are shared by all the cores, although per-core counters may provide more information. Figure 3-7 draws the contents of the four private lists and the private group table, when we extend the previous memory sequence (Figure 3-6) with three additional requests, A2, C1, and A1.

**Local/remote reuse counters**. The local counter of a group is incremented when a request falls into the respective group in the local private stack. In this example, only the last request, A1, encounters a local hit, and in this case, the local counter of the second group is incremented.

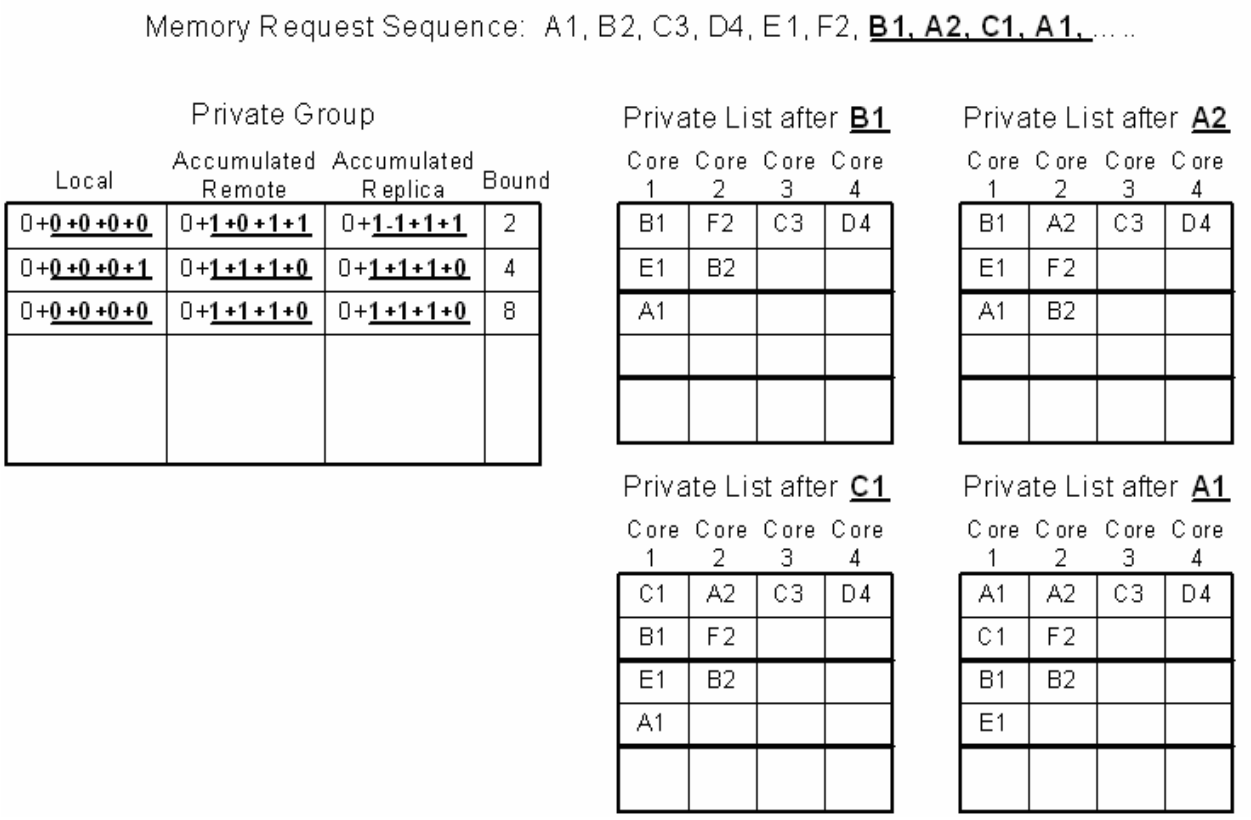


Figure 3-7. Example operations of the global stack for private caches.

After the simulation, the sum of all local counters from group 1 to group *m* represents the total number of local hits for private caches with exactly *m* groups.

Counting the remote hits is a little tricky, since a remote hit may only happen when a reference is a local miss. For example, assume that a request is in the third group of the local stack; meanwhile, the minimum group id of all the remote groups where this address appears is the second. When the private cache size is only large enough to contain the first group, neither a local nor a remote hit happens. If the cache contains exactly two groups, the request is a remote hit. Finally, if the cache is extended to the third group or larger, it is a local hit. Formally, if an address is present in the local group *L* and the minimum remote group that contains the block is *R*, the access can be a remote hit only if the cache size is within the range from group *R* to *L-1*. We increment the remote counters for groups *R* to *L-1* (*R* <= *L-1*). Note that after the simulation,

the remote counter *m* is the number of remote hits for a cache with exactly *m* groups. To differentiate them from the local counters, we call them *accumulated* remote counters.

In the example, the first highlighted request, B1, encounters a local miss, but a remote hit to B2 in the first group. We accumulate the remote counters for all the groups. The second request, A2, is also a local miss, but a remote hit to A1 in the second group. The remote counter of the first group remains unchanged, while the counters are incremented for all the remaining groups. Similar to B1, all the remote counters are incremented for C1. Finally, the last request, A1, is a local hit in the second group and is also a remote hit to A2 in the first group. In this case, only the remote counter of the first group is incremented since A1 is considered as a local hit if the cache size extends to more than the first group.

**Measuring replica**. The effective cache size is an important factor for shared and private cache comparisons [8], [24], [81], [20]. The single-pass stack simulation counts each block replication as a replica for calculating the effective cache size along the simulation. Similar to the remote hit case, we use accumulated replica counters. As shown in Figure 3-7, the first highlighted request, B1, creates a replica in the first group, as well as any larger groups because of the presence of B2. The second highlighted request, A2, does not create a new replica in the first group. But it does create a new replica in the second group because of A1. Meanwhile, A2 pushes B2 out of the first group, thus reduces a replica in the first group. This new replica applies to all the larger groups too. Note that the addition of B2 in the second group does not alter the replica counter for group 2, since the replica was already counted when B2 was first referenced. Similar to B1, the third highlighted request, C1, creates a replica to all the groups. Lastly, the reference, A1, extends a replica of A into the first group because of A2. The counters for the remaining groups stay the same.

**Handling memory writes**. In private caches, memory writes may cause invalidations to all the replicas. During the stack simulation, write invalidations create holes in the private stacks where the replicas are located. These holes will be filled later when the adjacent block is pushed down from a more-recently-used position by a new request. No block will be pushed out of a group when a hole exists in the group. To accurately maintain the reuse counters in the private group table, each group records the total number of holes for each core. The number of holes is initialized to the respective group size, and is decremented whenever a valid block joins the group. The hole-count for each group avoids searching for existing holes.

### 3.3 Evaluation and Validation Methodology

We simulate an 8-core CMP system. The global stack runs behind the L1 caches and simulates every L1 misses, essentially replacing the role of L2 caches. During simulations, stack distances and other related statistics are collected as described in the above section. Each group contains 256 blocks (16KB), and we simulate 1024 groups (16MB maximum). The results of the single-pass stack simulation are used to derive the performance of shared or private caches with various cache sizes and the sharing mechanisms for understanding the accessibility vs. capacity tradeoff in CMP caches.

The results from the stack simulation are verified against execution-driven simulations, where detailed cache models with proper access latencies are inserted. In the detailed execution-driven simulation, we assume the shared L2 has eight banks, with one local and seven remote determined by the least-significant three bits of the block address. The total shared cache sizes are 1, 2, 4, 8, and 16MB. For the private L2, we model both local and remote accesses. The MOESI coherence protocol is implemented to maintain data coherence among the private L2s. Accordingly, we simulate 128, 256, 512, 1024, 2048KB private caches. For comparison, we use the hit/miss information and average memory access times to approximate the execution time

behavior because the single-pass stack simulation cannot provide IPCs. We use three multithreaded commercial workloads, OLTP, Apache, and SPECjbb.

The accuracy of the CMP memory performance projection can be assessed from two different angles, the accuracy of predicting individual performance metrics, and the accuracy of predicting general cache behavior. By verifying the results against the execution-driven simulation, we demonstrate that the stack simulation can accurately predict cache hits and misses for the targeted L2 cache organizations, and more importantly, it can precisely project the sharing and replication behavior of the CMP caches.

One inherent weakness of stack simulation is its inability to insert accurate timing delays for variable L2 cache sizes. The fluctuation in memory delays may alter the sequence of memory accesses among multiple processors. We try a simple approach to insert memory delays based on a single discrete cache size. In the stack simulation, we inserted memory delays based on five cache sizes 1MB, 2MB, 4MB, 8MB, and 16MB,denoted as stack-1, stack-2, stack-4, stack-8, and stack-16 respectively. An off-chip cache miss latency is charged if the reuse distance is longer than the selected discrete cache size.

## 3.4 Evaluation and Validation Results

### 3.4.1 Hits/Misses for Shared and Private L2 Caches

Figure 3-8 shows the projected and real miss rates for shared caches, where “real” represents the results from individual execution-driven simulations. In general, the stack results follow the execution-driven results closely. For OLTP, stack-2 shows only about 5-6% average error. For Apache and SPECjbb, the difference among different delay insertions is less apparent. The stack results predict the miss ratios with about 2-6% error, except for Apache with a small 1MB cache.

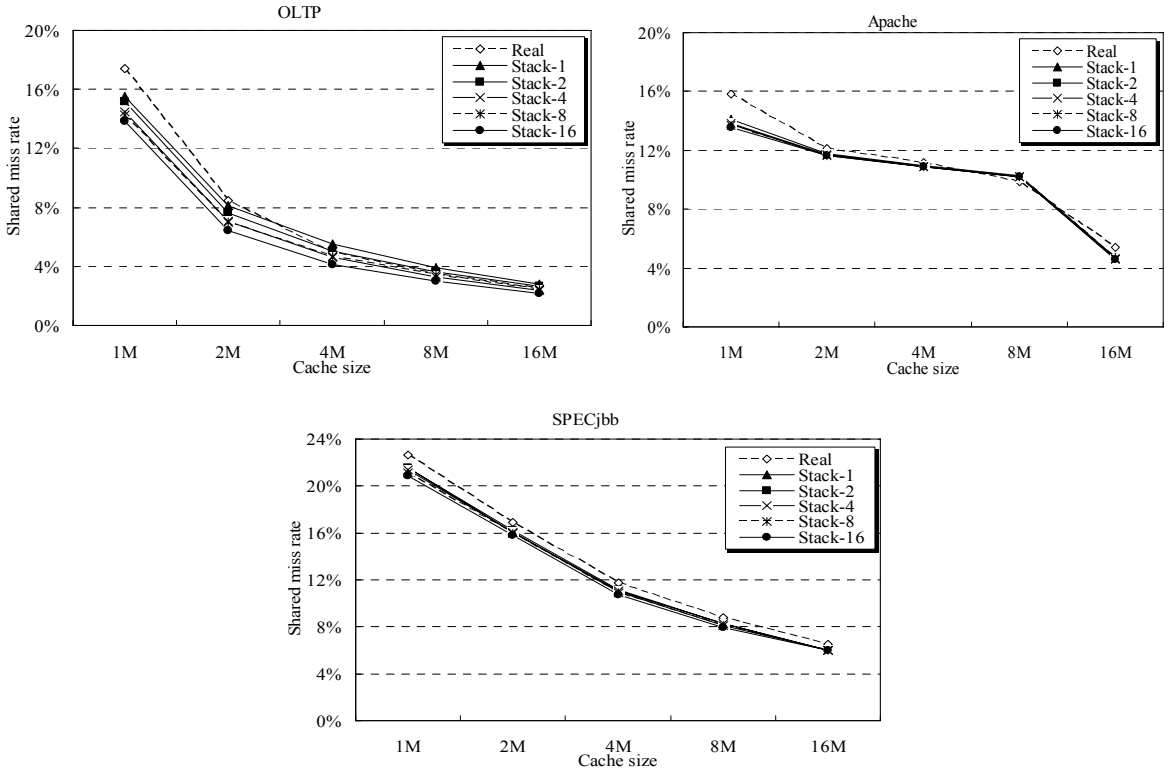


Figure 3-8. Verification of miss ratios from global stack simulation for shared caches.

Two major factors affect the accuracy of the stack results.

One is cache associativity. Since we use a fully-associative stack to simulate a 16-way cache, the stack simulation usually underestimates the real miss rates. This effect is more apparent when the cache size is small, due to more conflict misses. The issue can be solved by more complicated set-associative stack simulations [53], [32]. For simplicity, we keep the stack fully-associative. More sensitivity studies also need to evaluate L2 caches with smaller set associativity.

The other factor is inaccurate delay insertions. For example, in the stack-1 simulation of OLTP, the cache miss latency is inserted whenever the reuse distance is longer than 1MB. Such a cache miss delay is inserted wrongly for caches larger than the 1MB. These extra delays for larger caches cause more OS interference and context switches that may lead to more cache misses. At 4MB cache size, the overestimate of cache misses due to the extra delay insertion

exceeds the underestimate due to the full associativity. The gap becomes wider with larger caches. On the other hand, the stack-16 simulation for smaller caches mistakenly inserts hit latency, instead of miss latency, for accesses with reuse distance from the corresponding cache size to 16MB, causing less OS interferences, thus less misses. In this case, both the full associativity and the delay insertion lead to underestimate of the real misses, which makes the stack-16 simulation the most inaccurate.

For private caches, Figure 3-9 shows the overall misses, the remote hits, and the average effective sizes. Note that the horizontal axis shows the size of a single core from 128KB to 2MB each. With eight cores, the total sizes of the private caches are comparable to the shared cache sizes in Figure 3-8.

We can make two important observations. First, comapring with the shared cache, the simulation results show that the overall L2 miss ratios are increased by 14.7%, 9.9%, 4.3%, 1.1%, and 0.5% for OLTP for the private cache sizes from 128KB to 1MB. For Apache and SPECjbb, the L2 miss ratios are increased by 11.8%, 4.4%, 1.1%, 1.0%, 2.2%, and 7.3%, 3.1%, 2.9%, 0.6%, 0.5%, respectively. Second, the estimated miss and remote hit rates from the stack simulation match closely to the results from the execution-driven simulations, with less than 10% margin of errors.

We also simulate the effective capacity for the private-cache cases. The effective cache size is the average over the entire simulation period. In general, the private cache reduces the cache capacity due to replicated and invalid cache entries. The effective capacity is reduced to 45-75% for the three workloads with various cache sizes. The estimated capacity from the stack simulation is almost identical to the result from the execution-driven simulation. Due to its higher accuracy, we use the stack-2 simulation in the following discussion.

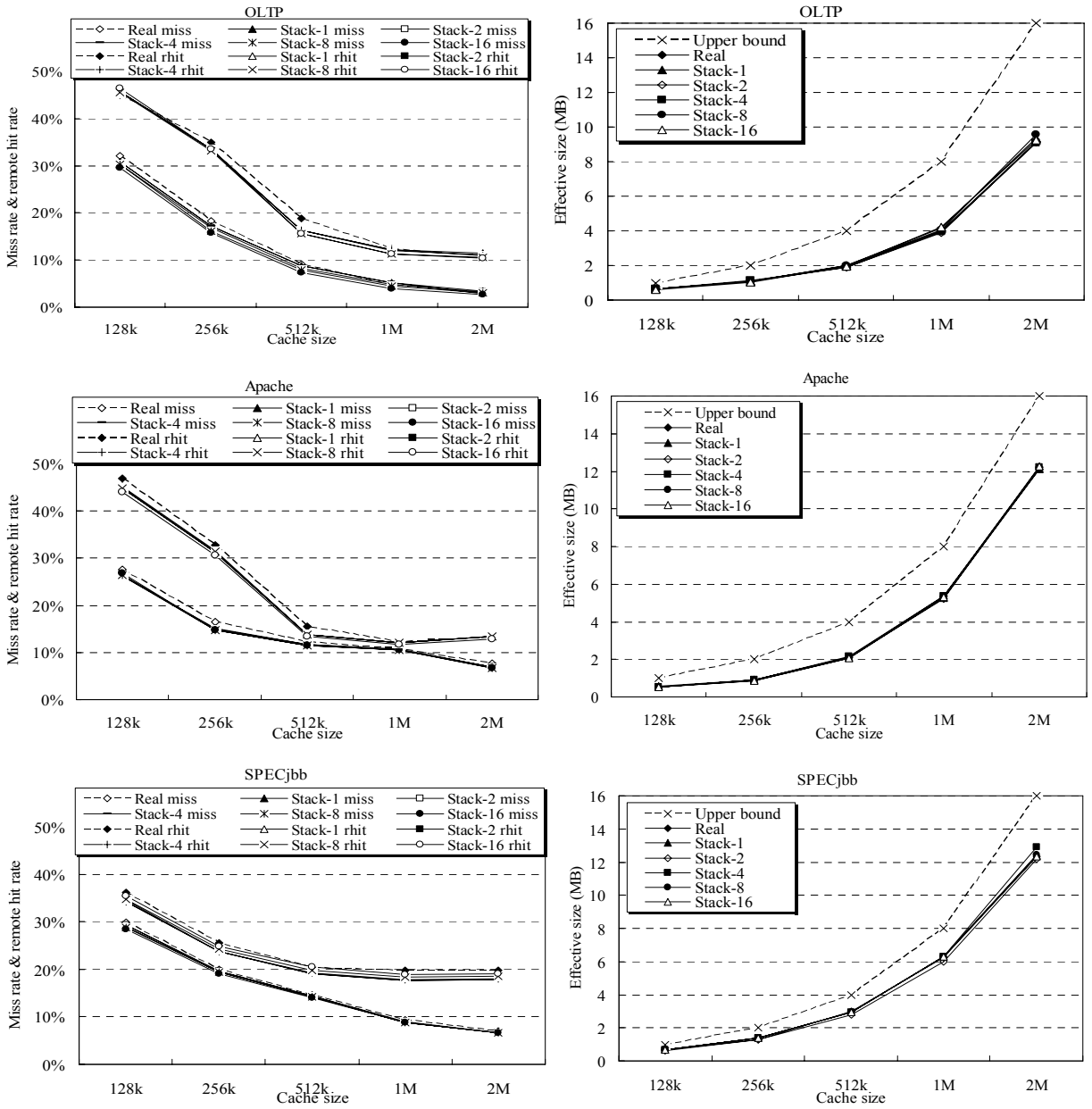


Figure 3-9. Verification of miss ratio, remote hit ratio and average effective size from global stack simulation for private caches.

### 3.4.2 Shared Caches with Replication

To balance accessibility and capacity, victim-replication [81] creates a dynamic L1 victim cache for each core in the local slice of the L2 to trade capacity for fast local access. In this section, we estimate the performance of a static victim-replication scheme. We allocate 0% to 50% of the L2 capacity as L1 victim caches with variable L2 sizes from 2MB to 8MB. For

performance comparison, we use the average memory access time, which is calculated based on the local hits to victim caches, the hits to shared portion of L2, and L2 misses.

The average memory access time of the static victim replication can be derived directly from the results of the stack simulation described in the previous sections. Assuming the inclusion property is enforced between the shared potion of the L2 and the victim potion plus the L1. Suppose the L1 and L2 sizes are denoted by *CL1*, and *CL2*, r is the percentage of the L2 allocated for the victim cache, and n is the number of the cores. Then, each victim-cache size is equal to *(r*CL2)/n*, and the remaining shared portion is equal to *(1-r)*CL2*. The average memory access time includes the following components. First, since the L1 and the victim cache are exclusive, the total hits to the victim cache can be estimated from the private stacks with the size of the L1 plus the size of the victim:*CL1+(r*CL2)/n*. Note that this estimation may not be precise due to the lack of the L1 hit information that alters the sequence in the stack. Second, the total number of L2 hits (including the victim portion) and L2 misses can be calculated from the shared stack with the size *(1-r)*CL2*. Finally, the hit to the shared portion of L2 can be calculated by subtracting the hits to the victim from the total L2 hits.

Figure 3-10 demonstrates the average L2 access time with static victim replication. Generally, large caches favor more replications. For a small 2MB L2, except that Apache has a slight performance gain at low replication levels, the average L2 access times increase with more replications. The optimal replication levels for OLTP are 12.5%, and 37.5% respectively for 4MB and 8MB L2. This general performance behavior with respect to data replication is consistent with what we have observed from the analytical model in section 3.1. However, the analytical model without cache invalidations should apply lower *L* for the optimal replication level.

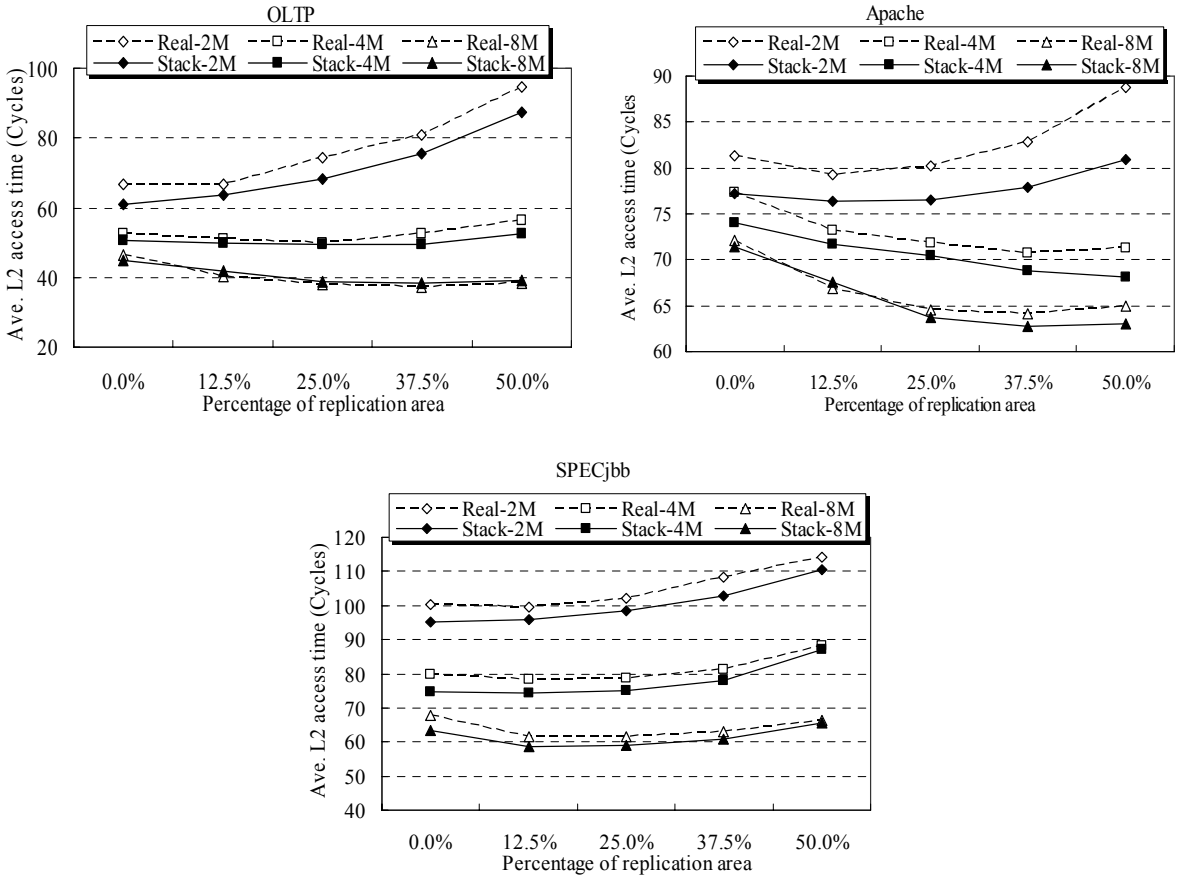


Figure 3-10. Verification of average L2 access time with different level of replication derived from global stack simulation for shared caches with replication.

For SPECjbb, 12.5% replication shows the best for both 4MB and 8MB L2. The figure for Apache shows that the performance is better with replications as large as 50% for 4MB, but 37.5% for 8MB L2s. The seemly contradiction comes from the fact that L2 misses start to reduce drastically around 8M caches, as demonstrated in Figure 3-8. We can also observe that the optimal replication levels match perfectly between the stack simulations and the execution-driven simulations. With respect to the average L2 access time, the stack results are within 2%-8% error margins.

### 3.4.3 Private Caches without Replication

Private caches sacrifice capacity for fast access time. It may be desirable to limit replications in the private caches. To understand the impact of the private L2 without replication,

we run a separate stack simulation in which the creation of a replica causes the invalidation of the original copy.

Figure 3-11 demonstrates the L2 access delays of the private caches without replication, shown as the ratio to those of the private caches with full replication. As expected, with small 128KB and 256KB private caches per core, the average L2 access times without replication are about 5-17% lower than those with full replication for all the three workloads. This is because the benefit of the increased capacity more than compensates the loss of local accesses.

With large 1MB or 2MB caches per core, the average L2 access time of the private caches without replication is 12-30% worse than the full-replication counterpart, suggesting that increasing local accesses is beneficial when enough L2 capacity is available. The stack simulation results follow this trend perfectly. They provide very accurate results with only 2-5% margin of error.

### 3.4.4 Simulation Time Comparison

The full-system Virtutech Simics 2.2 simulator [50] to simulate an 8-core CMP system with Linux 9.0 and x86 ISA is running on Intel Xeon 3.2 GHz 2-way SMP. The simulation time of each stack or execution-driven simulation is measured on a dedicated system without other interference. A timer was inserted at the beginning and the end of each run to calculate the total execution time.

In the single-pass stack simulation, each stack is partitioned into 16KB groups with a total of 1024 groups for the 16MB cache. This small 16KB groups are necessary in order to study shared caches with variable percentage of replication areas as shown in Figure 3-10. The stack simulation time can be further reduced for cache organizations that only require a few large groups.

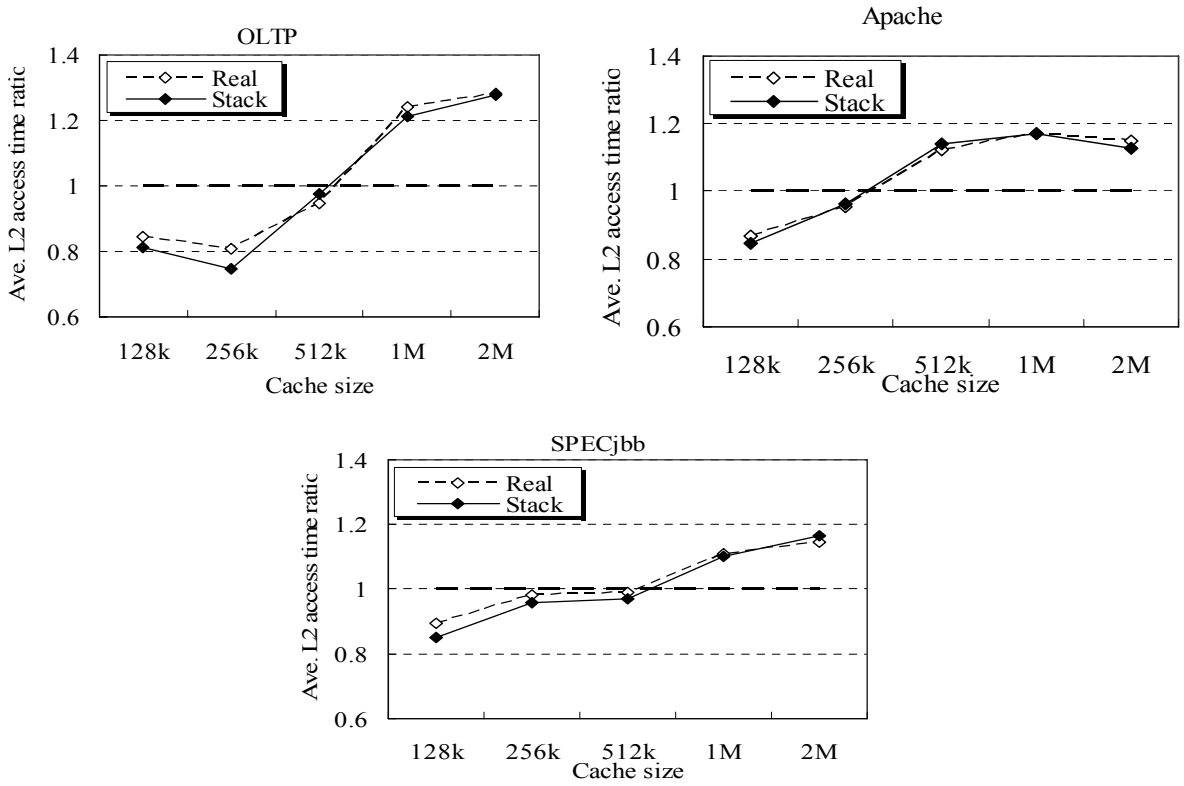


Figure 3-11. Verification of average L2 access time ratio from global stack simulation for private caches without replication.

Table 3-1 summaries the simulation times for the stack and the execution-driven simulations to obtain the results above. For each workload, two stack simulations are needed. One run is for producing the results for shared caches, private caches, and shared caches with replication, and the other run is for the private L2 without replication. In execution-driven simulations, it requires a separate run for each cache size resulting in five runs for each cache organization. In studying the shared cache with replication, five separate runs are needed for each cache size in order to simulate five different replication percentages. No separate stack simulation is required for the shared cache with replication. Similarly, no separate execution-driven simulation is needed for shared caches with 0% area for data replication. Therefore we have 20 runs for the shared with replication for execution-driven simulations. The total number of simulation runs is also summarized in Table 3-1.

Table 3-1. Simulation time comparison of global stack and execution-driven simulation (in Minutes)

| Measurements | Workload | Stack | Execution-Driven |
|---|---|---|---|
| Shared / Private (Section 3.4.1) | OLTP | 1 Run: 835 | (5+5) Runs: 6252 |
| | Apache | 1 Run: 901 | (5+5) Runs: 6319 |
| | SPECjbb | 1 Run: 582 | (5+5) Runs: 4220 |
| Shared with replication (Section 3.4.2) | OLTP | 0 Run: 0 | 20 Runs: 11976 |
| | Apache | 0 Run: 0 | 20 Runs: 12211 |
| | SPECjbb | 0 Run: 0 | 20 Runs: 8210 |
| Private no replication (Section 3.4.3) | OLTP | 1 Run: 872 | 5 Runs: 3257 |
| | Apache | 1 Run: 948 | 5 Runs: 3372 |
| | SPECjbb | 1 Run: 613 | 5 Runs: 2199 |
| Total | | 4751 | 58016 |

The total stack simulation time is measured about 4751 minutes, while the execution-driven simulation takes 58016 minutes, a factor over 12 times. This gap can be much wider if more cache organizations and sizes are studied and simulated.

## 3.5 Summary

In this chapter, we developed an abstract model for understanding the general performance behavior of data replication in CMP caches. The model showed that data replication could degrade cache performance without a sufficiently large capacity. We then used the global stack simulation for more detailed study on the issue of balancing accessibility and capacity for on-chip storage space on CMPs. With the stack simulation, we can explore a wide-spectrum of the cache design space in a single simulation pass. We simulated the schemes of shared caches, private caches and private caches without replication with various cache sizes directly by global stack simulation. We also deduce the performance data of shared caches with replication by the shared and private cache results. We verified the stack simulation results with execution-driven simulations using commercial multithreaded workloads. We showed that the single-pass stack simulation can characterize the CMP cache performance with high accuracy (about only 2% - 9% error magins) and significant less simulation time (only 8 %). Our results proved that the

effectiveness of various techniques to optimize the CMP on-chip storage is closely related to the total L2 size.

# CHAPTER 4
# DIRECTORY LOOKASIDE TABLE: ENABLING SCALABLE, LOW-CONFLICT CMP CACHE COHERENCE DIRECTORY

Directory cache coherence mechanism is one of the most important choices for building scalable CMPs. The design of a sparse coherence directory for future CMPs with many cores presents new challenges. With a typical set-associative sparse directory, the hot-set conflict at the directory tends to worsen when many cores compete in each individual set, unless the set associativity is dramatically increased. In order to maintain precise cache information, the set-conflict causes inadvertent cache invalidations. Thus, an important technical issue is to avoid the hot-set conflicts at the coherence directory with small set associativity, small directory space and high efficiency.

We develop a set-associative directory with an augmented directory lookaside table to allow displacing directory entries from their primary sets for solving the hot-set conflicts. The proposed CMP coherence directory offers three unique contributions. First, while none of the existing cache coherence mechanisms are efficient enough when the number of cores becomes large, the proposed CMP coherence directory provides a low cost design with a small directory size and low set associativity. Second, although the size of the coherence directory matches the total number of CMP cache blocks, the topological difference between the coherence directory and all cache modules creates conflicts in individual sets of the coherence directory and causes inadvertent invalidations. The DLT is introduced to reconcile the mismatch between the two CMP components. In addition, the unique design of the DLT has its own independent utility in that it can be applied to other set-associative cache organizations for alleviating hot-set conflicts. In particular, it has advantages over other multiple-hash-function-based schemes, such as the skewed associative cache [64], [14]. Third, unlike the memory-based coherence directory where each memory block has a single directory entry along with the presence bits indicating where the

block is located, the proposed CMP directory keeps a separate record for every copy of the same cached block along with the core ID. Multiple hits to a block can occur in a directory lookup, which returns multiple core IDs without expensive presence bits.

Performance evaluations have demonstrated the significant performance improvement of the DLT-enhanced directory over the traditional set-associative or skewed associative directories. Augmented with a DLT that allows up to one quarter of the cache blocks to be displaced from their primary sets in the set-associative directory, up to 10% improvement in execution time is achievable. More importantly, such an improvement is within 98% of what an ideal coherence directory can accomplish.

In the following sections of this chapter, we first show the problem that a limited set-associative CMP coherence directory may have big performance impact due to inadvertent cache invalidations. We then propose our enhancement of the directory, directory lookaside table. This is followed by detailed performance evalutions.

### 4.1 Impact on Limited CMP Coherence Directory

In this section, we demonstrate the severity of cache invalidation due to hot-set conflicts at the coherence directory if the directory has small set-associativity. Each copy of a cached block occupies a directory entry that records the block address and the ID of the core where the block is located. A block must be removed from the cache when its corresponding entry is replaced from the CMP directory. Three multithreaded workloads, OLTP, Apache, SPECjbb, and two multiprogrammed workloads, SPEC2000 and SPEC2006, were used for this study. These workloads ran on a Simics based whole-system simulation environment. In these simulations, we assume a CMP with eight cores, and each core has a private 1MB, 8-way, L2 cache. The simulated CMP directory with different set associativities can record a total of 8MB cache blocks. Detailed descriptions of the simulation will be given in Section 4.3.

Figure 4-1 shows the average of the total valid cache blocks over a long simulation period using a CMP coherence directory with various set associativities. Given eight cores, each with an 8-way set-associative L2 cache, the 64-way directory (*Set-full*) can accommodate all cache blocks without causing any extra invalidation. The small percentage of invalid blocks for the 64-way directory comes from cache coherence invalidations due to data sharing, OS interference, thread migrations, etc. on multiple cores.

The severity of cache invalidations because of set conflicts in the directory is very evident in the cases of smaller set associativities. In general, whenever the set associativity is reduced by half, the total valid cache blocks are reduced by 4-9% for all five workloads. Using OLTP as an example, the valid blocks are reduced from 93% to 87%, 82% and 75% as the set associativity is reduced from 64 ways to 32, 16, and 8 ways, respectively. The gap between the 64-way and 8-way directories indicates that, on average, about 18% of the total cached blocks are invalidated due to insufficient associativity in the 8-way coherence directory. This severe decrease in valid blocks will reduce the local cache hits and increase the overall CMP cache misses.

To further demonstrate the effect of hot-set conflicts in a directory with small set-associativity, we also simulated an 8-way set-associative directory with twice the number of sets, capable of recording the states for a cache size of 16 MB (denoted as *2x-8way*). We can observe that a significant gap in the number of valid blocks still exists between the *2x-8way* and the 64-way directories. For OLTP, the 32-way directory can also out-perform the *2x-8way* directory.

To completely avoid extra cache invalidations, a 64-way directory is needed here. Consider a future CMP with 64 cores and 16-way private L2 caches, an expensive and power hungry 1024-way directory is needed to eliminate all extra cache invalidations. This full-associativity directory is essentially the same as maintaining and searching all individual cache directories.

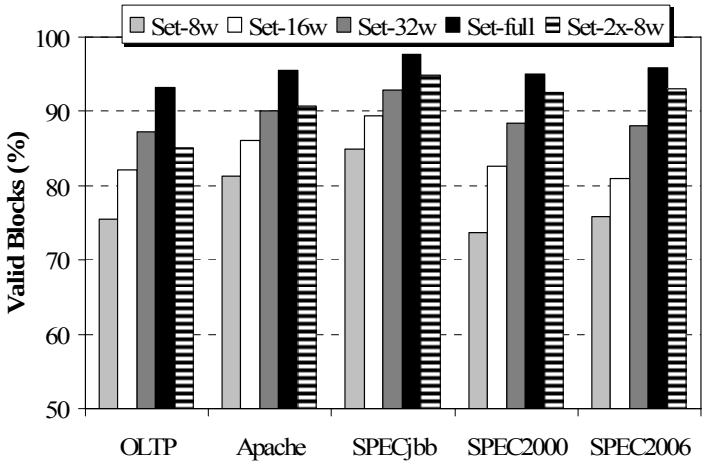

Figure 4-1. Valid cache blocks in CMP directories with various set-associativity.

### 4.2 A New CMP Coherence Directory

To solve or reduce the hot-set conflicts, we propose to displace the replaced directory entries to the empty slots in the directory.

Figure 4-2 illustrates the basic organization of a CMP coherence directory enhanced with a Directory Lookaside Table (DLT), referred collectively as the *DLT-dir*. The directory part, called the main directory, is set-associative. Each entry in the main directory records a cached block with its address tag, a core ID, a MOESI coherence state, a valid bit, and a bit indicating whether the recorded block has been displaced from its primary set.

The DLT is organized as a linear array in which each entry can establish a link to a displaced block in the main directory. Each DLT entry consists of a pointer, the index bits of the displaced block in the main directory, and a 'use' bit indicating if the DLT entry has a valid pointer. In addition, a set-empty bit array is used to indicate whether the corresponding set in the main directory has a free entry for accommodating any displaced block away from the block's primary set. Note that, different from the memory-based directory, the *DLT-dir* serves only as a coherence directory without any associated data array.

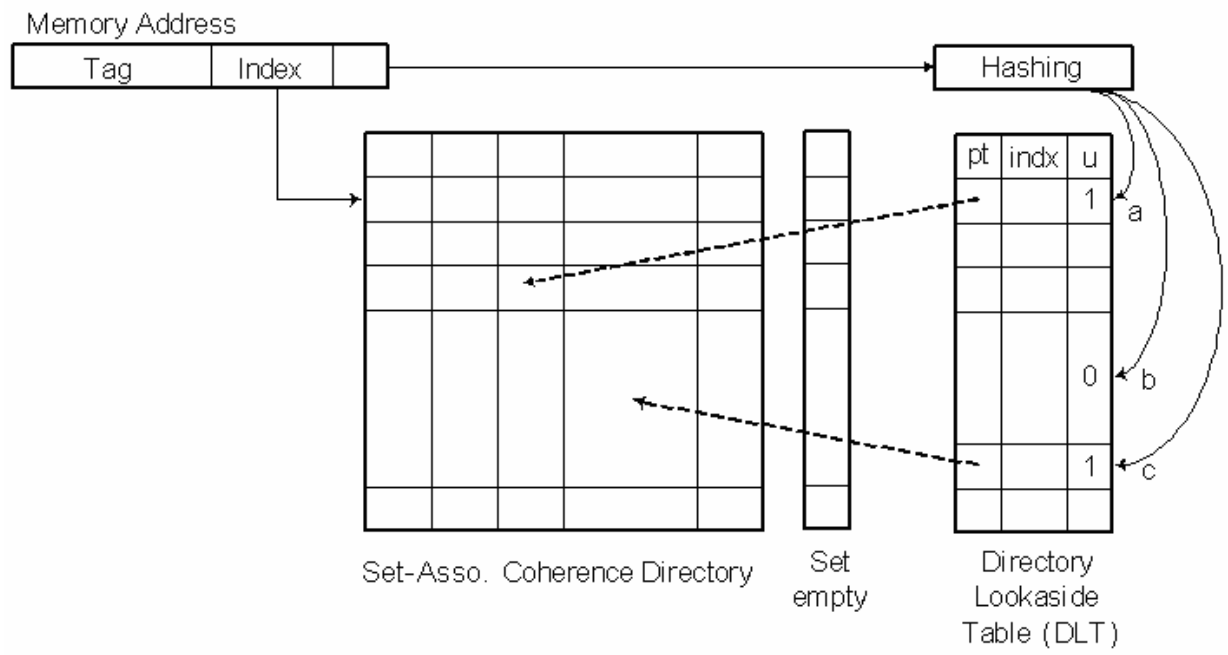


Figure 4-2. A CMP coherence directory with a multiple-hashing DLT.

Inspired by a similar idea in the skewed associative cache [64], [14], a set of hash functions are used to index the DLT; the purpose is to reduce the conflict at the DLT. When a block is to be displaced in the main directory due to a conflict, the hash functions are applied to the block address to obtain multiple locations in the DLT. Some of these locations may have already been used to point to some other displaced blocks in the main directory. If there exist unused DLT locations among the hashed ones and if there also exists a free slot in the main directory, then the displaced block is moved to the free directory slot and an unused DLT location is selected to point to the displaced block. Since this paper is not aiming at inventing new hash functions, we borrow the skewed function family reported in [14]. Let $\sigma$ be the one-position circular shift on n index bits, a cache block at memory address $A = A_3 2^{c+2n} + A_2 2^{c+n} + A_1 2^c + A_0$ can be mapped to the following m locations: $f_0(A) = A_1 \oplus A_2$, $f_1(A) = \sigma(A_1) \oplus A_2$, $f_2(A) = \sigma^2(A_1) \oplus A_2$, …, and $f_{m-1}(A) = \sigma^{m-1}(A_1) \oplus A_2$, where $m \le n$.

In the illustrated example of Figure 4-2, a block address is hashed to three locations, a, b, and c in the DLT based on the skewed functions [14]. Indicated by the 'use' bit, location a or c contains a valid pointer that points to a displaced block in the main directory, while location b is unused. The index bits of the main directory are attached in the DLT for the displaced blocks for two purposes. First, such a scheme saves the main directory space by not including any index bits in the address tag at each directory entry. Note that these index bits are needed only for the displaced blocks. Instead of allocating space at every directory entry for storing the index bits, no such space is allocated at all in the main directory and the index bits of the displaced blocks are stored in the DLT. Second, the index bits in the DLT can be used to filter out unnecessary access to the main directory. An access is granted only when the index bits match with that of the requested address. In the example of Figure 4-2, assume both a and c's indexes match with that of the requested address, then access to the main directory is initiated. The address tags of the two main directory entries pointed by location a and c are compared against the address tag of the request. When a tag match occurs, a displaced block is found. Note that although DLT-dir requires additional directory access beyond the primary sets, our evaluation shows that a majority (about 97-99%) of this secondary access can be filtered out using the index bits stored in the DLT.

In the attempt to relocate a block evicted from its primary set to another directory entry, suppose a free DLT slot, say b, is found for the block. A free slot in the main directory must also be identified. The set-empty bit array is maintained for this purpose. The corresponding set-empty bit is set whenever a free directory slot appears. A quick scan to the set-empty bit array returns a set with at least one free slot. The displaced block is then stored in the free slot in the main directory and that location is recorded in entry b in the DLT.

When a block is removed from a cache module either due to eviction or invalidation, the block must also be removed from the DLT-dir. If the block is recorded in its primary set, all that is needed is to turn the valid bit off. If the block is displaced, it will be found through a DLT lookup. Both the main directory entry for the block and the corresponding DLT entry are freed.

A directory entry that holds a displaced block can also be replaced by a newly referenced block. Given that the index is unavailable in the main directory for the displaced block, normally a backward pointer is needed from the directory entry that holds the displaced block to the corresponding DLT entry, in order to free the DLT entry. However, it is expensive to add a backward pointer in the main directory. Alternatively, the DLT entry can be searched for determining the DLT entry that points to the location of the displaced block in the main directory. If each DLT entry is allowed to point to any directory location, then a fully associative search of the DLT is required to locate a given location in the main directory. To reduce the cost of searching the DLT, one can impose restriction on the DLT-to-directory mapping so that, for a given directory location, only a small subset of the DLT entries can potentially point to it and need to be examined. For instance, consider a DLT whose total number of entries is one-quarter of that of the directory entries, which will be shown to be sufficiently large. In the most restrictive DLT-to-directory mapping, each DLT entry is allowed to point to one of only four fixed locations in the main directory. Although the need of DLT search is minimal, such a restrictive mapping could lead to severe hot-set-like conflicts during the displacement of a block, because the block can only be displaced to a small number of potential locations in the directory: The block is first mapped to several DLT entries (by multiple hash functions), each of which in turn can point to one of a small number of directory entries. The result is a reduced chance of finding a free directory entry for the displaced block. In a less restrictive design, any set-

associative mapping can be instituted such that each DLT entry is limited to certain collection of sets in the main directory. The set-associative mapping allows fast search in the DLT for a directory location with minimum reduction on the chance of finding free slots in the main directory. We will evaluate the performance of this design in Section 4.4.

In comparison with other multiple-hash-function-based directories or caches, e.g., the skewed associative directory, the multiple-hashing DLT has its unique advantages. Since the DLT is used only to keep track of the free slots and displaced blocks in the main directory, its size, counted in number of entries, is considerably smaller than the total number of entries in the main directory. Suppose the directory has a total of 1000 entries, then the DLT may have 250 entries. Suppose the directory contains 100 displaced blocks and 100 free slots at some point. Then, the directory has 10% of free entries but the DLT has (250-100)/250 = 60% of free entries. When the same hash function family is used in both the skewed associative directory and the DLT, the chance of finding a free entry in the DLT is much higher (0.9993 vs. 0.5695 if eight uniform random hash functions are used). Once a free DLT entry is found, finding a free entry in the directory is ensured by searching the set-empty bit array (assuming the DLT-to-directory mapping is unrestricted). We will demonstrate the performance advantage of this unique property.

The detailed operations of the DLT-dir are summarized as follows.

When a requested block is absent from the local cache, a search of the DLT-dir is carried out for locating the requested block in other cache modules. When the block is found in its primary set of the main directory and/or in other sets through the DLT lookup, proper coherence actions are performed to fetch and/or invalidate the block from other caches. The block with the requesting core ID is inserted into the MRU position in the primary set of the main directory.

This newly inserted block may lead to the following sequence of actions. First, the block is always inserted into a free entry in the primary set if one exists. Otherwise, it replaces a displaced block residing in the primary set; this design is intended to limit the total number of displaced blocks. In this case, the previously established pointer in the DLT to the displaced block must be freed. If no displaced block exists in the primary set, the LRU block is replaced. In either case, the replaced block will undergo a displacement attempt through the DLT. If no free space is found in either the main directory or the DLT, the replaced block is evicted from the directory and invalidated in the cache module.

To displace a block, an unused entry in the DLT must be selected through the multiple hash functions. In addition, the set-empty bit array is checked for selecting a free slot in the main directory to which the selected DLT entry can be mapped. Each corresponding bit in the set-empty array is updated every time the respective set is searched.

A miss in the CMP caches is encountered if the block cannot be found in the DLT-dir. The corresponding block must be fetched from a lower-level memory in the memory hierarchy. The update of the DLT-dir for the newly fetched block is the same as when the block is found in other CMP cache modules.

When a write request hits a block in the shared state in the local cache, an upgrade request is sent to the DLT-dir. The requested block with the core ID must exist either in the primary set or in other sets linked through the DLT. The requested block can exist in more than one entry in the main directory since the shared block may also be in other cores' caches. In response to the upgrade request, all other copies of the block must be invalidated in the respective caches and their corresponding directory entries must be freed. Replacement of any block in a cache module

must be accompanied by a notification to the directory for freeing the corresponding directory and DLT entries.

### 4.3 Evaluation Methodology

We use Simics to evaluate an 8-core out-of-order x86 chip multiprocessor. We develop detailed cycle-by-cycle event-driven cache, directory and interconnection models. Each core has its own L1 instruction and data caches, and an inclusive private L2 cache. Every core has its own command-address-response bus and data bus connecting itself with all the directory banks. The MOESI coherence protocol is applied to maintain cache coherence through the directory. Each core has two outgoing request queues to the directory, a miss-request queue and a replacement-notification queue, and an outgoing response queue for sending the response to the directory. It also has an incoming request queue for handling request from the directory and an incoming response queue. Each bank of the directory maintains five corresponding queues for each core to buffer request/reponse from/to each core. The simulator keeps track of the states of each bus and directory bank, as well as all the queues. The timing delays of directory access, bus transmission, and queue conflicts are carefully modeled. We assume the DLT access can be fully overlapped with the main directory access. However, the overall access latency may vary based on the number of displaced blocks that need to be checked. Besides the main directory access latency of 6 cycles, we assume that 3 additional cycles are consumed for each access to a displaced block after the access is issued from the DLT. Table 4-1 summarizes the directory related simulation parameters besides the general parameters narrated in Chapter 1.

For this study, we use three multithreaded commercial workloads, OLTP (Online Transaction Processing), Apache (static web server), and SPECjbb (java server), and two multiprogrammed workloads with applications from SPEC2000 and SPEC2006.

Table 4-1. Directory-related simulation parameters

| Parameter | Description |
|---|---|
| CMP | 8-core, 1M private L2 cache each core |
| Main directory | 1/2/4/8 Banks, 128K entries, 8 way |
| Queue size | 8-entry request/response queues to/from each core |
| DLT table | 1/2/4/8 Banks, 8K/16K/32K entries |
| DLT mapping | each DLT entry maps to 8/16/32/64/128 directory sets |
| Directory latency | 6 cycles for primary set, each displaced block for 3 additional cycles |
| Remote latency | 52 cycles without contention, 4 hops |
| Cmd/Data bus | 8B, bidirectional, 32GB/s, 6-cycle propagation latency |

Table 4-2. Space requirement for the seven directory organizations

| Directory | 8 Cores | Overhead | 64 Cores | Overhead |
|---|---|---|---|---|
| Set-8w | 4587520 | 1 | 36700160 | 1 |
| Set-8w-64v | 4671552 | 1.02 | 37372416 | 1.02 |
| Skew-8w | 6160384 | 1.34 | 49283072 | 1.34 |
| Set-10w-1/4 | 5734400 | 1.25 | 45875200 | 1.25 |
| Set-8w-p | 5242880 | 1.14 | 97517568 | 2.66 |
| DLT-8w-1/4 | 5439488 | 1.18 | 43515904 | 1.18 |
| Set-full | 4980736 | 1.09 | 39845888 | 1.09 |

We evaluated seven directory organizations: the 8-way set-associative (*Set-8w*), the 8-way set-associative with 64-block fully-associative victim buffer (*Set-8w-64v*), the 8-way skew associative (*Skew-8w*), the 10-way set-associative with 25% additional directory size (*Set-10w-1/4*), the 8-way set-associative with presence bits (*Set-8w-p*), the 8-way set associative with a DLT of 25% of total cache entries using 8 hashing functions (*DLT-8w-1/4*), and the full 64-way set-associative (*Set-full*) directory. The *Set-full* represents the ideal case where no directory conflict will occur. The *Set-10w-1/4* is included because it adds one-quarter extra directory space that matches the DLT entries in the *DLT-8w-1/4*. The *Set-10w-1/4* possesses an extra advantage because the set-associativity is increased from 8-way to 10-way. In our simulations, the DLT-dir is partitioned into four banks based on the low-order two bits in the block address to allow for sufficient directory bandwidth. Multiple-banked directory also displays interesting effects on the DLT-dir conflict. Detailed evaluation will be given in Section 4.4. All results are the average from all four banks. The total number of bits and the normalized space requirement relative to

that of the *Set-8w* for the seven directories are shown in Table 4-2 for an 8-core CMP and a 64-core CMP. The skewed associative directory (*Skew-8w*), requires the index bits as a part of the address tag, and hence needs the largest space. The space requirement for the directory with presence bits (*Set-8w-p*) goes up much faster than others when the number of cores increases (e.g. from 1.14 to 2.66).

## 4.4 Performance Result

In this section, we show performance evaluation results of the seven CMP coherence directory organizations. The cache hit/miss ratios, the average valid blocks, the IPC improvement, and sensitivity to the DLT design parameters are presented.

### 4.4.1 Valid Block and Hit/Miss Comparison

Figure 4-3 shows the percentage of valid blocks for the seven directories averaged over the entire simulation period. We can make a few important observations.

First, the proposed *DLT-8w-1/4* is far superior to any other directory organizations, except for the ideal *Set-full*. The *DLT-8w-1/4* can retain almost all cached blocks using a DLT whose total number of entries is equal to one-quarter of the total number of the cache blocks. Among the five workloads, only OLTP shows noticeable invalidations under *DLT-8w-1/4*. This is because, with intensive data and instruction sharing, OLTP experiences more set conflicts due to replication of shared blocks. Our simulation results reveal that over 40% of the cached blocks are replicas in OLTP.

Second, set-associative directories other than the one with the full 64-way perform poorly. For instance, in the *Set-8w* directory, about 18%, 14%, 13%, 21%, and 20% of the cached blocks are unnecessarily invalidated for the respective five workloads, due to hot-set conflicts in the directory. The *Set-10w-1/4* directory improves the number of valid blocks at the cost of 25% additional directory entries and higher associativity. But, its deficit is still substantial.

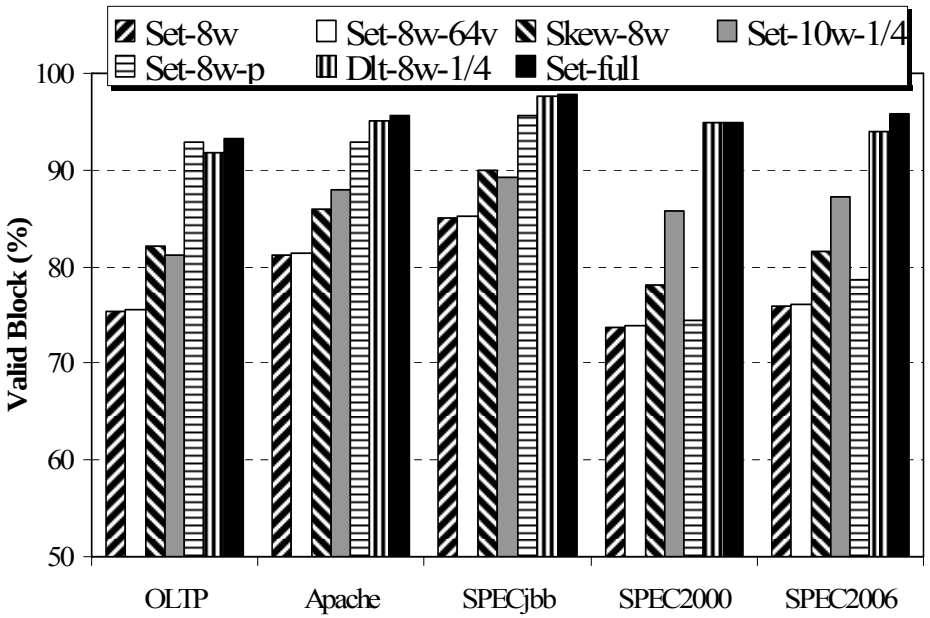


Figure 4-3. Valid cache blocks for simulated cache coherence directories.

Third, very little advantage is shown when the *Set-8w* is furnished with a 64-block fully-associative victim buffer. Apparently, the buffer is too small to hold a sufficient number of conflicting blocks.

Fourth, the skewed associative directory (*Skew-8w*) does alleviate set conflict. But, it is still far from being able to retain all the valid blocks. Since multiple-hashing is applied to the entire directory, the chance of finding a free slot in the main directory is diluted by the non-displaced blocks located in the primary sets. In contrast, the DLT only contains the displaced blocks; hence the chance of finding a free slot in the DLT is much higher (See the sample calculation near the end of Section 4.2 for detail.).

Lastly, the *Set-8w-p* works well with multithreaded workloads, but performs poorly with multiprogrammed workloads. By combining (duplicated) blocks with the same address into one entry in the directory, the presence-bit-based implementation saves directory entries, and hence, alleviates set conflicts for multithreaded workloads. However, keeping the presence bits becomes increasingly space-inefficient as the number of cores increases. In addition, for

multiprogrammed workloads, there is little data sharing; the advantages of having the expensive presence bits no longer exist.

The total valid blocks determine the overall cache hits and misses. In Figure 4-4, extra L1 hits, L2 local hits, L2 remote hits, and L2 misses of the seven directory schemes, normalized with respect to the total L2 references of the *Set-8w*, are displayed for each workload. Note that the inadvertent invalidation due to set conflicts at the directory also invalidates the blocks located in the L1 caches for maintaining the L1/L2 inclusion property. Therefore, more L1 misses are encountered for the directory schemes that cause more invalidation.

Compared with the *Set-8w*, there are extra L1 hits and fewer L2 references for all other directory schemes. Among the workloads, SPECjbb under the *DLT-8w-1/4* sees about 8% increase in the L1 hits. The *DLT-8w-1/4* shows clear advantages over the other directory schemes.

Besides additional L1 hits and fewer L2 misses, the biggest gain for using the *DLT-dir* comes from the increase of the local L2 hits. Recall that to avoid the expensive presence bits, each copy of a block occupies a separate directory entry with a unique core ID. Consequently, inadvertent invalidation of a cached block caused by insufficient directory space may not turn a L2 local hit to the block into an extra L2 cache miss. Instead, it is likely that a local L2 hit to the block results in a remote L2 hit since not all copies of the block are invalidated.

This is the key reason for more remote L2 hits in the directory schemes that produce more invalidation for the three multithreaded workloads. The difference among the directory schemes is not as significant for SPEC2000 and SPEC2006 since there is little data sharing among the multiprogrammed workload.

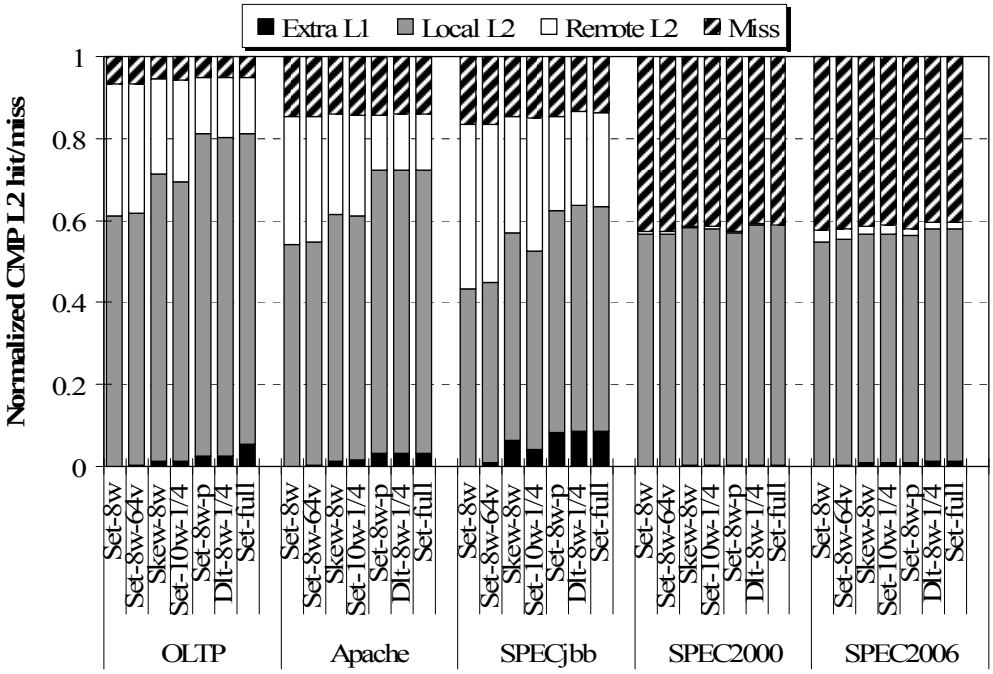


Figure 4-4. Cache hit/miss and invalidation for simulated cache coherence directories.

Figure 4-5 plots the distribution of four types of requests to the directory: instruction fetch (IFetch), data read (Dread), data write hit to a shared-state block (Upgrade), and data write miss (DWrite). Within each request type, the results are further broken down into four categories based on the directory search results: hit only to the main directory, hit only to the DLT, hit to both the main directory and the DLT, and miss at both the main directory and the DLT (which becomes a CMP cache miss).

A few interesting observations can be made. First, there are very few requests that find the blocks only through the DLT for all request types in all five workloads. Given the fact that a displaced block is used to be at the LRU position of its primary set, the chance of its reuse is not high, unless other copies of the same block are also in the primary set such that the request is likely targeting for the blocks in the primary set. Therefore, a hit to the DLT is usually accompanied by one or more hits to the main directory. This is especially true for IFetch in the three multithreaded workloads with more sharing of read-only instructions among multiple cores.

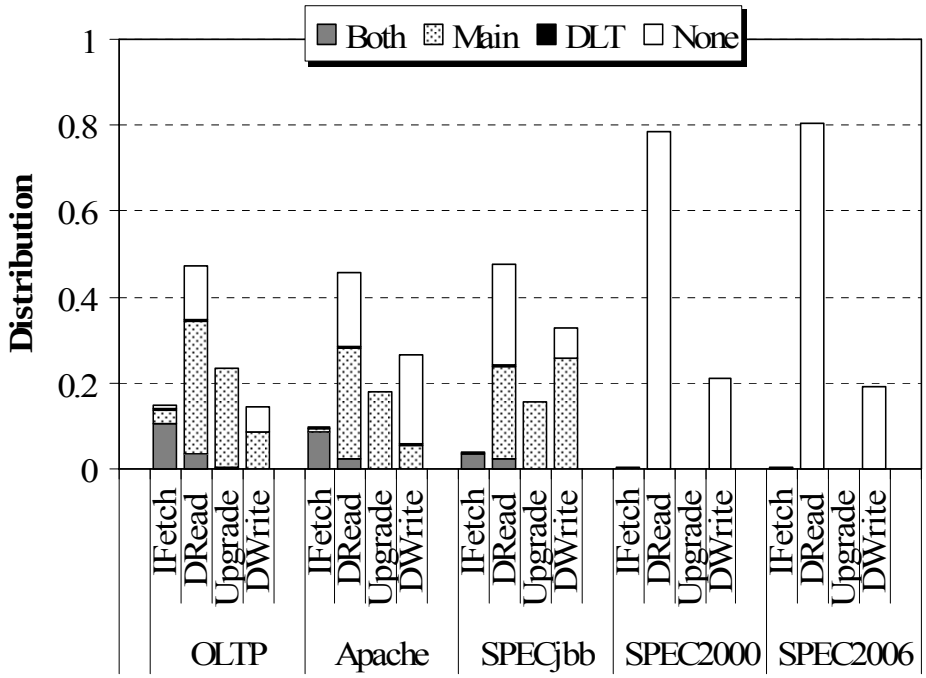


Figure 4-5. Distribution of directory hits to main directory and DLT.

A good percentage of instruction blocks are displaced from their primary sets due to heavy conflicts. A small percentage of DRead also finds the requested blocks in both places. Since neither IFetch nor DRead requires locating all copies of the block, it is not harmful to look up the main directory and the DLT sequentially to save power.

Second, it is observed that very few displaced blocks are encountered by the Upgrade or DWrite requests. Detailed analysis of the results indicates that widely shared blocks do exist, but they are mostly read-only blocks. This is demonstrated by the fact that the average number of sharers for DRead is about 6 when DRead hits both the main directory and the DLT. But, for Upgrade and DWrite, the number of sharers is less than 3, a small enough number so that the three copies of the block can be kept in the primary set most of the time. This explains why such blocks are not typically found through the DLT. Nevertheless, parallel searching of the directory and DLT is still desirable because the required acknowledgement can be sent back to the requesting core faster.

Third, for the two multiprogrammed workloads, there are almost no Ifetch requests to the DLT-dir, revealing the small footprint of the instructions. Since there is no data sharing, both DRead and DWrite are always misses. Moreover, there are no Upgrade requests because the blocks are in the E-state.

### 4.4.2 DLT Sensitivity Studies

Two key DLT design parameters are its size and number of hash functions. In Figure 4-6, we vary the DLT sizes among 1/16, 1/8, and 1/4 of the total number of the cache blocks. We also evaluate the difference between using 8 or 12 hash functions for accessing the DLT. As observed earlier from Figure 4-3, the average number of valid blocks drops by 13-21% when the Set-8w is used instead of the Set-full. To reduce the number of invalidated blocks, the DLT must be capable of capturing at least these percentages of blocks and allow them to be displaced from their primary sets. Therefore, we can observe significant improvement in the average number of valid blocks as the DLT size increases from 1/16 to 1/8 and to 1/4 of the total number of cache blocks for all five workloads.

The impact of the number of hash functions, on the other hand, is not as obvious. Eight hash functions are generally sufficient for finding an unused entry in the DLT when the DLT has sufficient size. We observe small improvement using 12 instead 8 hash functions. For example, the total number of valid blocks increases from 90.3% to 91.7% for OLTP with 12 hash functions.

The index bits, which are needed for displaced blocks, are recorded in the DLT to save space in the main directory. These index bits can also be used to filter the access to the main directory for the displaced blocks. An access is necessary only when the index bits match with the requested block.

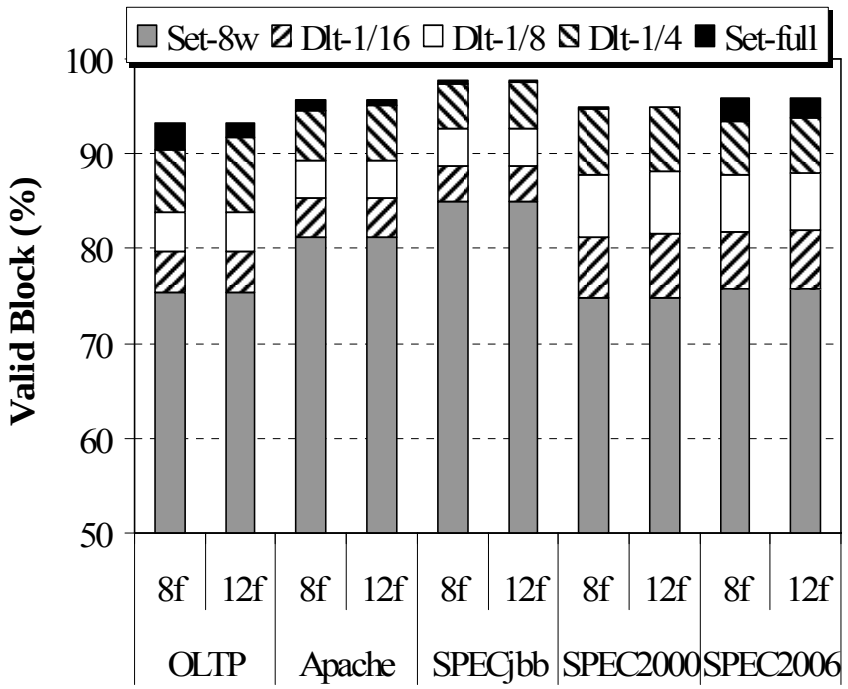


Figure 4-6. Sensitivity study on DLT size and number of hashing functions.

Figure 4-7 shows the advantage of filtering for the four workloads with the *DLT-8w-1/4* directory. Beside the 12 index bits, we also experiment with using additional 1, 2, or 3 bits for filtering purpose. Figure 4-7 (A) shows the false-positive rate, Figure 4-7 (B) illustrates the total traffic that can be filtered out. The false positive is defined as an index-bit match, but a failure to find the block in the main directory.

We observe that by recording one additional bit beyond the necessary index bits, the false-positive is almost completely eliminated for all three multithreaded workloads. For SPEC2000 and SPEC2006, however, additional 3 bits are necessary to reduce the false-positive rate to a negligible level. Furthermore, by recording one additional bit beyond the index bits, the total additional traffic to the main directory is reduced down to only about 0.2-2.8% of the total request traffic arriving at the DLT. With such high percentages of filtering, a majority of CMP cache misses or upgrades can be identified earlier without searching again the main directory for possible displaced blocks.

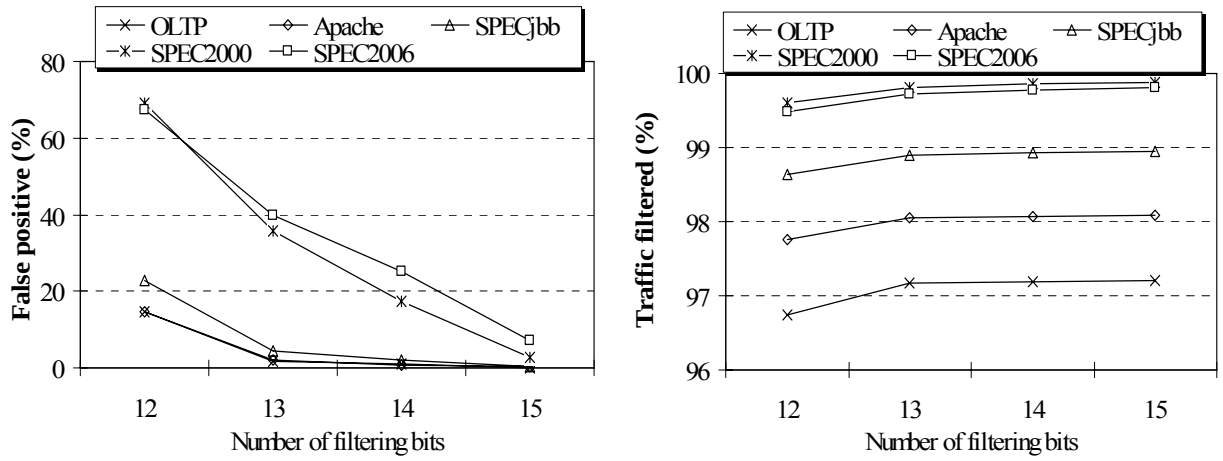


Figure 4-7. Effects of filtering directory searches by extra index bits.

The DLT is partitioned into multiple banks for enabling simultaneously DLT access from multiple cores. Meanwhile, for performing fast inverse search of the DLT given a main directory entry, the DLT is organized in a way that each DLT entry can only point to one set among a fixed sub-collection of all sets in the main directory.

Figure 4-8 shows the sensitivity studies with respect to these two parameters for OLTP and SPEC2000. The X-axis represents the number of fixed sets in the main directory that each DLT entry can point to; the Y-axis is the normalized invalidation with respect to the total invalidations in a reference configuration where the directory has only one bank and the DLT uses a 8-set mapping.

Several interesting observations can be made. First, banking also helps reducing inadvertent invalidations. The main reason is that with smaller banks, the same DLT-to-directory mapping covers a larger percentage of the main directory. For example, for an 8-way directory corresponding to an 8MB cache with a 64-byte block size, there are 16K sets for a 1-bank directory, but only 4K sets per bank for a 4-bank directory. Therefore, if each DLT entry can be mapped to 32 sets, it covers 1/512 of the directory entries for the 1-bank case. However, with the same mapping, a DLT entry can cover 1/128 of the directory entries for the 4-bank directory.

The more the coverage, the higher the chance will be for finding a free slot in the main directory for displacement. The curves corresponding to the same directory covering percentage by the each DLT entry are also drawn with broken lines in the figure.

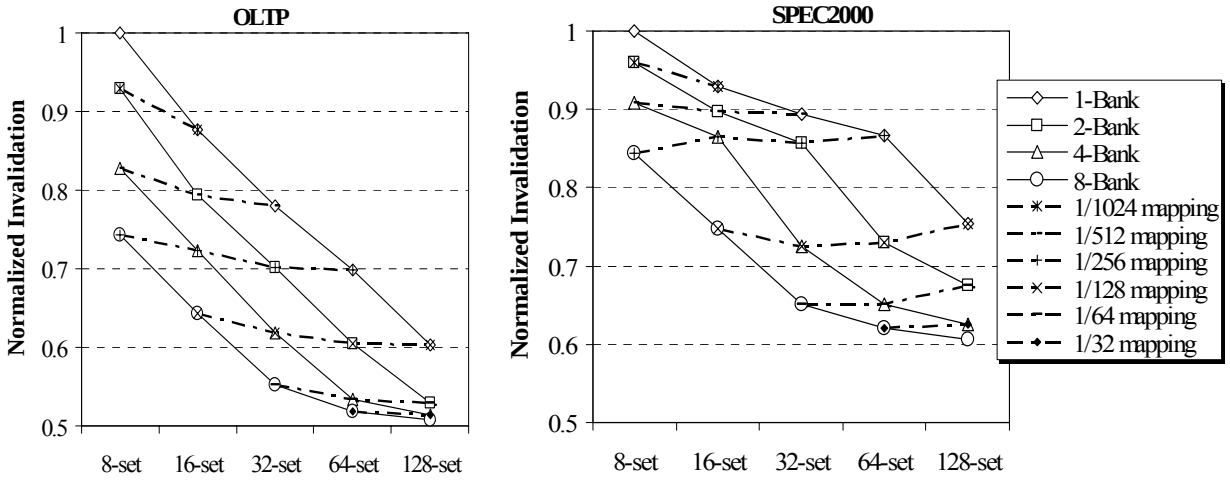


Figure 4-8. Normalized invalidation with banked DLT and restricted mapping from DLT to directory.

Second, banking creates another level of conflicts because the distribution of the cached blocks to each bank may not be even. The physical bank prevents the DLT-based block displacement from crossing the bank boundary. This effect is very evident in OLTP. With the same covering percentage, the 1-bank directory performs noticeably better than the 2-bank directory, which in turn is better than the 4-bank and the 8-bank directories. The bank conflict is not as obvious for SPEC2000 due to its mixed applications. Simple index randomization technique can be applied to alleviate the bank conflicts. But, further discussions are omitted due to space limitation.

Third, the constrained DLT-to-directory mapping does affect the number of invalidations substantially. But, the reduction of invalidations starts diminishing when doubling the mapping freedom from 1/64 to 1/32. In the early simulation, we used a 4-bank directory with a 32-set mapping, where each DLT entry can be mapped to 1/128 banked directory space, in order to

achieve a balance between minimizing the amount of invalidations and the cost of searching the DLT entries.

### 4.4.3 Execution Time Improvement

The execution times for the seven directory schemes are compared in Figure 4-9. The figure shows the normalized execution times with respect to the *Set-8w* directory for each workload. For the *DLT-8w-1/4* directory, the execution time improvement is about 10%, 5%, 9%, 8%, and 8% over the Set-8w directory. More importantly, the improvement of the *DLT-8w-1/4* is about 98% of what the full 64-way directory (*Set-full*) can achieve for all five workloads. In the case of the more expensive skewed associative directory, the time saved is only about 20-35% of what the *DLT-8w-1/4* can save for the three multithreaded workloads. Finally, the *Set-8w-p* reduces the execution time of the multithreaded workloads, but does little for the multiprogrammed ones.

## 4.5 Summary

In this chapter, we describe an efficient cache coherence mechanism for future CMPs with many cores and many cache modules. We argue in favor of a directory-based approach because the snooping-bus-based approach lacks scalability due to its broadcasting nature. However, the design of a low-cost coherence directory with small size and small set-associativity for future CMPs must handle the hot-set conflicts at the directory that lead to unnecessary block invalidations at the cache modules. In a typical set-associative directory, the hot-set conflict tends to become worse because many cores compete in each individual set with uneven distribution. The central issue is to reconcile the topology difference between the set-associative coherence directory and the CMP caches.

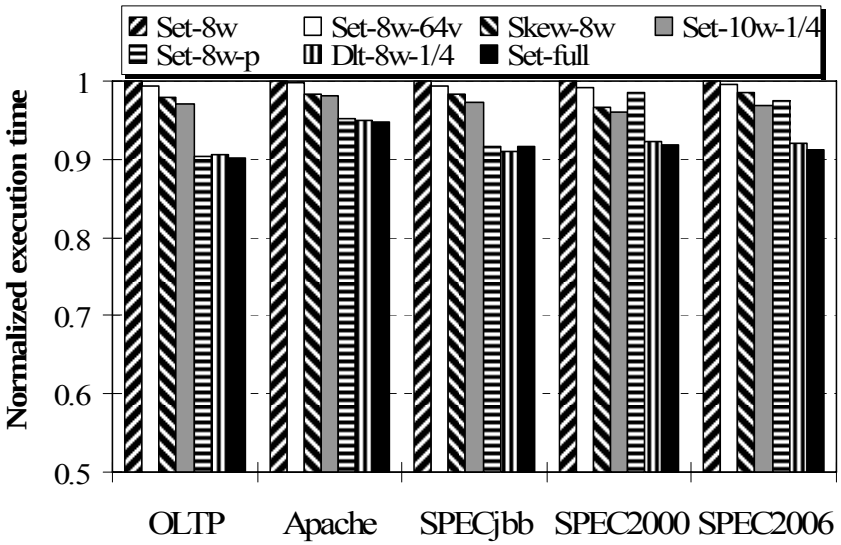


Figure 4-9. Normalized execution time for simulated cache coherence directories.

The proposed *DLT-dir* accomplishes just that with small space requirement and high efficiency. The hot-set conflict is alleviated by allowing blocks to be displaced from their primary sets. A new DLT is introduced to keep track of the displaced blocks by maintaining a pointer to the displaced block in the main directory. The DLT is accessed by applying multiple hash functions to the requested block address to reduce the DLT's own conflicts. Performance evaluation has confirmed the advantage of the *DLT-dir* over other conventional set associative directories or the skewed-associative directory for conflict avoidance. In particular, the *DLT-dir* with a size equal to one quarter of the total number of the directory blocks achieves up to 10% faster execution time in comparison with a traditional 8-way set-associative directory.

# CHAPTER 5
# DISSERTATION SUMMARY

Chip multiprocessors (CMPs) are becoming ubiquitous in all computing domains. As the number of cores increases, tremendous pressures will be exerted on the memory hierarchy system to supply the instructions and data in a timely fashion to sustain increasing chip-level IPCs. In this dissertation, we develop three works about bridging the ever-increasing CPU memory performance gap.

In the first part of this dissertation, an accurate and low-overhead data prefetching on CMPs based on a unique observation of coterminous group (CG) and coterminous locality has been developed. A coterminous group is a group of off-chip memory accesses with the same reuse distance. Coterminous locality is defined as when a member in the coterminous group is accessed, the other members will be likely to be accessed in the near future. In particular, the order of nearby references in a CG follows exactly the same order that these references appeared last time, even though they may be irregular. The proposed prefetcher uses CG history to trigger prefetches when a member in a group is re-referenced. It overcomes challenges of the existing correlation-based or stream-based prefetchers, including low prefetch accuracy, lack of timeliness, and large history. The accurate CG-prefetcher is especially appealing for CMPs, where cache contentions and memory access demands are escalated. Evaluations based on various workload mixes have demonstrated significant advantages of the CG-prefetcher over other existing prefetching schemes on CMPs, with about 10% of IPC improvement with much less extra traffic.

As many techniques have been proposed for optimizing on-chip storage space in CMPs, the second part of the dissertation is to propose an analytical model and a global stack simulation to fast project the performance of the tradeoff between the capacity and access latency in CMP

caches. The purposed analytical model can fast estimate the general performance behavior of data replication in CMP caches. The model has showed that data replication could degrade cache performance without a sufficiently large capacity. The global stack simulation has been proposed for more detailed study on the issue of balancing accessibility and capacity for on-chip storage space on CMPs. With the stack simulation, a wide-spectrum of the cache design space can be explored in a single simulation pass. We have simulated the schemes of shared/private caches, and shared caches with replication of various cache sizes. We also have verified the stack simulation results with execution-driven simulations using commercial multithreaded workloads and showed that the single-pass stack simulation can characterize the CMP cache performance with high accuracy. Only 2%-9% of error margin is observed. Our results have proved that the effectiveness of various techniques to optimize the CMP on-chip storage is closely related to the total L2 size. More importantly, our global stack simulation consumes only 8% of the simulation time of execution-driven simulations.

In this third part of the dissertation, we have described an efficient cache coherence mechanism for future CMPs with many cores and many cache modules. We favor a directory-based approach because the snooping-bus-based approach lacks scalability due to its broadcasting nature. However, the design of a low-cost coherence directory with small size and small set-associativity for future CMPs must handle the hot-set conflicts at the directory that lead to unnecessary block invalidations at the cache modules. In a typical set-associative directory, the hot-set conflict tends to become worse because many cores compete in each individual set with uneven distribution. The central issue is to reconcile the topology difference between the set-associative coherence directory and the CMP caches. The proposed DLT-dir accomplishes that just with small space requirement and high efficiency. The hot-set conflict is alleviated by

allowing blocks to be displaced from their primary sets. A new DLT is introduced to keep track of the displaced blocks by maintaining a pointer to the displaced block in the main directory. The DLT is accessed by applying multiple hash functions to the requested block address to reduce the DLT's own conflicts. Performance evaluation has confirmed the advantage of the DLT-dir over other conventional set associative directories or the skewed-associative directory for conflict avoidance. In particular, the DLT-dir with a size equal to one quarter of the total number of the directory blocks achieves up to 10% faster execution time in comparison with a traditional 8-way set-associative directory.

LIST OF REFERENCES


[1] M. E. Acacio, "A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems Vol. 16 (1)*, pp. 67-79, Jan. 2005

[2] Advanced Micro Devices, "AMD Demonstrates Dual Core Leadership," *http://www.amd.com*, 2004.

[3] AMD Quad-Core, *http://multicore.amd.com/us-en/quadcore/*

[4] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Int'l Symp. Computer Architecture*, pp. 280-289, May 1988.

[5] A. Agarwal, M. Horowitz, and J. Hennessy, "An Analytical Cache Model," *ACM Trans. on Computer Systems, Vol. 7, No. 2*, pp. 184-215, May 1989.

[6] A. Agarwal and S. D. Pudar, "Column-Associative Caches: a Technique for Reducing the Miss Rate of Direct-Mapped Caches," *Proc. 29th Int'l Symp. on Computer Architecture*, pp. 179-190, May 1993.

[7] L. Barroso et al, "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Int'l Symp. on Computer Architecture*, pp. 165-175, June 2000.

[8] B. Beckmann and D. Wood, "Managing Wire Delay in Large Chip-multiprocessor Caches," *Proc. 37th Int'l Symp. on Microarchitecture*, Dec. 2004.

[9] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive Selective Replication for CMP Caches," *Proc. 39th Int'l Symp. on Microarchitecture*, Dec. 2006.

[10] B. T. Bennett and V. J. Kruskal, "LRU Stack Processing," *IBM journal of R & D*, pp. 353-357, July 1975.

[11] E. Berg and E. Hagersten, "StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis," *Proc. 2004 Int'l Symp. on Performance Analysis of Systems and Software*, March 2004.

[12] E. Berg, H. Zeffer, and E. Hagersten, "A Statistical Multiprocessor Cache Model," *Proc. 2006 Int'l Symp. on Performance Analysis of Systems and Software*, March 2006.

[13] B. Black et al, "Die Stacking (3D) Microarchitecture," *Proc. 39th Int'l Symp. on Microarchitecture*, pp. 469-479, Dec. 2006.

[14] F. Bodin and A. Seznec, "Skewed Associativity Improves Performance and Enhances Predictability," *IEEE Trans. on Computers, 46(5)*, pp. 530-544, May 1997.

[15] S. Borkar, "Microarchitecture and Design Challenges for Gigascale Integration," *Proc. 37th Int'l Symp. on Microarchitecture, 1st Keynote*, pp. 3-3, Dec. 2004.

[16] P. Bose, "Chip-Level Microarchitecture Trends," *IEEE Micro, Vol 24(2)*, pp. 5-5, Mar-Apr. 2004.

[17] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. on Computers, c-27(12)*, pp. 1112-1118, Dec. 1978.

[18] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-Based Cache Coherence in Large-Scale Multiprocessors," *Computer 23, 6*, pp. 49-58, Jun. 1990.

[19] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture", *Proc. 11th Int'l Symp. on High Performance Computer Architecture*, pp. 340-351, Feb. 2005.

[20] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," *Proc. 33rd Int'l Symp. on Computer Architecture*, June 2006.

[21] B. Chazelle and Kilian et al, "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables," *Proc. 15th Annual ACM-SIAM Symp. on Discrete Algorithms*, Jan. 2004.

[22] T. Chen and J. Baer, "Reducing Memory Latency via Non-blocking and Prefetching Caches," *Proc. of Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 51-61, Oct. 1992.

[23] T. M. Chilimbi and M. Hirzel, "Dynamic Hot Data Stream Prefetching for General-purpose Programs," *Proc. SIGPLAN '02 Conference on PLDI*, pp. 199-209, June 2002.

[24] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing Replication, Communication, and Capacity Allocation in CMPs," *Proc. 32nd Int'l Symp. on Computer Architecture*, June 2005.

[25] D. E. Culler, J. P. Singh, and A. Gupta, "Parallel Computer Architecture: A Hardware/Software Approach," *Morgan Kaufmann Publishers Inc.*, 1999.

[26] H. Dybdahl and P. Stenstrom, "An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors," *Proc. 13th Int'l Symp. on High Performance Computer Architecture*, Feb 2007.

[27] G. Edwards, S. Devadas, and L. Rudolph, "Analytical Cache Models with Applications to Cache Partitioning," *Proc. 15th Int'l Conf. on Supercomputing*, pp. 1-12, June 2001.

[28] B. Fraguela, R. Doallo, and E. Zapata, "Automatic Analytical Modeling for the Estimation of Cache Misses," *Proc. 1999 Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sep. 1999.

[29] J. D. Gilbert, S. H. Hunt, D. Gunadi, and G. Srinivasa, "Niagara2: A Highly-Threaded Server-on-A-Chip," *Proc. 18th HotChips Symp*, Aug. 2006.

[30] A. Gupta, W. Weber, and T. Mowry. "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proc. Int'l Conf. ICPP '90*, pp. 312-321, Aug. 1990.

[31] J. Hasan and S. Cadambi et al. "Chisel: A Storage-efficient, Collision-free Hash-based Network Processing Architecture," *Proc. 33th Int'l Symp. on Computer Architecture*, pp. 203-215, June 2006.

[32] M. Hill and J. Smith, "Evaluating Associativity in CPU Caches", *IEEE Trans. on Computers*, pp. 1612-1630, Dec. 1989.

[33] Z. Hu, M. Martonosi, and S. Kaxiras, "TCP: Tag Correlating Prefetchers," *Proc. 9th Int'l Symp. on HPCA*, pp 317-326, Feb. 2003.

[34] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA Substrate for Flexible CMP Cache Sharing," *Proc. 19th Int'l Conf. on Supercomputing*, June 2005.

[35] Intel Core Duo Processor: The Next Leap in Microprocessor Architecture, *Technology@Intel Magazine*, Feb. 2006.

[36] Intel Core 2 Quad Processors, *http://www.intel.com/products/processor/core2quad/index.htm*

[37] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *Proc. 26th Int'l Symp. on Computer architecture*, pp. 252-263, June 1997.

[38] N. P. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers," *Proc. 17th Int'l Symp. on Computer Architecture*, pp 364-373, May 1990.

[39] R. Kalla, B. Sinharoy, and J. Tendler, "IBM POWER5 Chip: A Dual-Core Multithreaded Processor," *IEEE Micro, Vol 24(2)*, Mar-Apr. 2004.

[40] S. Kapil, "UltraSPARC Gemini: Dual CPU Processor," *Proc. 15th HotChips Symp.*, Aug. 2003.

[41] C. Kim, D. Burger, and S. Keckler, "An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches," *Proc 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[42] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," *Proc. 2004 Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2004.

[43] Y. H. Kim, M. D. Hill, and D. A. Wood, "Implementing Stack Simulation for Highly-associative Memories," *Proc. 1991 SIGMETRICS conf. on Measurement and Modeling of Computer Systems*, pp. 212-213, May 1991.

[44] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro, Vol 26(3)*, pp. 10-23, May-June 2006.

[45] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way Multithreaded SPARC Processor," *Proc. 16th HotChips Symp.*, Aug. 2004.

[46] S. Lacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective Stream-Based and Execution-Based Data Prefetching," *Proc. 19th Int'l Conf. on Supercomputing*, pp 1-11, June 2004.

[47] A. Lai, C. Fide, and B. Falsafi, "Dead-block Prediction & Dead-block Correlating Prefetchers," *Proc. 28th Int'l Symp. on Computer Architecture*, pp 144-154, July 2001.

[48] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir, "Design and Management of 3D Chip Multiprocessors Using Network-in-Memory," *Proc. 33rd Int'l Symp. on Computer Architecture*, June 2006.

[49] C. Liu, A. Sivasubramaniam, and M. Kandemir, "Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs," *Proc. 10th Int'l Symp. on High Performance Computer Architecture*, pp. 176-185, Feb. 2004.

[50] P. S. Magnusson et al, "Simics: A Full System Simulation Platform," *IEEE Computer*, Feb. 2002.

[51] M. R. Marty and M. D. Hill, "Virtual Hierarchies to Support Server Consolidation," *Proc. 34th Int'l Symp. on Computer Architecture*, June 2007.

[52] Matlab, *http://www.mathworks.com/products/matlab/*.

[53] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation Techniques and Storage Hierarchies," *IBM Systems Journal, 9*, pp. 78-117, 1970.

[54] K. Nesbit and J. Smith, "Data Cache Prefetching Using a Global History Buffer," *Proc. 10th Int'l Symp. on High Performance Computer Architecture*, pp 96-105, Feb. 2004.

[55] B. O'Krafka and A. Newton, "An Empirical Evaluation of Two Memory-Efficient Directory Methods," *Proc. 17th Int'l Symp. Computer Architecture*, pp. 138-147, May 1990.

[56] K. Olukotun et al, "The Case for a Single-Chip Multiprocessor," *Proc. 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[57] Open Source Development Labs Database Test 2, *http://www.osdl.org /lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-2/*.

[58] J. K. Peir, Y. Lee, and W. W. Hsu, "Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology," *Proc. 8th Int'l Conf. on Architectural Support for Programming Language and Operating Systems*, pp. 240-250, Oct. 1998.

[59] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The V-Way Cache: Demand-Based Associativity via Global Replacement," *Proc. 32nd Int'l Symp. on Computer Architecture*, June 2005.

[60] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," *Proc. 33rd Int'l Symp. on Microarchitecture*, Dec. 2006.

[61] N. Rafique, W. Lim, and M. Thottethodi, "Architectural Support for Operating System Driven CMP Cache Management," *Proc. 2006 Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2006.

[62] S. Sair and M. Charney, "Memory Behavior of the SPEC-2000 Benchmark Suit," *Technical Report, IBM Corp.*, Oct. 2000.

[63] A. Saulsbury, F. Dahlgren, and P. Stenstrom, "Recency-based TLB preloading," *Proc. 27th Int'l Symp. on Computer architecture*, pp. 117-127, May 2000.

[64] A. Seznec, "A Case for Two-Way Skewed-Associative Cache," *Proc. 20th Int'l Symp. on Computer Architecture*, pp. 169-178, May 1993.

[65] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. on Architecture Support for Programming Language and Operating Systems*, pp 45–57, Oct. 2002.

[66] G. Sohi, "Single-Chip Multiprocessors: The Next Wave of Computer Architecture Innovation," *Proc. 37th Int'l Symp. on Microarchitecture, 2nd Keynote*, pp. 143-143, Dec. 2004.

[67] Y. Solihin, J. Lee, and J. Torrellas, "Using a User-level Memory Thread for Correlation Prefetching," *Proc. 29th Int'l Symp. on Computer Architecture*, pp.171-182, May 2002.

[68] M. Spjuth, M. Karlsson, and E. Hagersten, "Skewed Caches from a Low-power Perspective," *Proc. 2nd Conf. on Computing frontiers 2005*, May 2005.

[69] L. Spracklen and S. Abraham, "Chip Multithreading: Opportunities and Challenges," *Proc. 11th Int'l Symp. on High Performance Computer Architecture*, pp. 248-252, Feb. 2005.

[70] L. Spracklen and Y. Chou, "Effective Instruction Pre-fetching in Chip Multiprocessors for Modern Commercial Applications," *Proc. 11th Int'l Symp. on High Performance Computer Architecture*, pp. 225-236, Feb. 2005.

[71] K. Strauss, X. Shen, and J. Torrellas, "Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors," *Proc. 33rd Int'l Symp. on Computer Architecture*, June 2006.

[72] R. A. Sugumar and S. G. Abraham, "Set-associative Cache Simulation using Generalized Binomial Trees," *ACM Trans. on Computer Systems, Vol. 13, No. 1*, pp. 32-56, Feb. 1995

[73] C. K. Tang, "Cache Design in the Tightly Coupled Multiprocessor System," *AFIPS Conference Proceedings, National Computer Conference*, pp 749-753, June 1976.

[74] S. P. Vanderwiel and D. J. Lilja, "Data Prefetch Mechanisms," *ACM Computing Surveys*, pp. 174-199, June 2000.

[75] S. Vangal et al., "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS," *IEEE International Solid-State Circuits Conference*, Feb. 2007.

[76] X. Vera and J. Xue, "Let's Study Whole-Program Cache Behavior Analytically," *Proc. 8th Int'l Symp. on High Performance Computer Architecture*, Feb. 2002.

[77] Z. Wang and D. Burger, et al., "Guided Region Prefetching: a Cooperative Hardware/Software Approach," *Proc. 30th Int'l Symp. on Computer Architecture*, pp. 388-398, June 2003.

[78] T. Wenisch and S. Somogyi, et al., "Temporal Streaming of Shared Memory," *Proc. 32nd Int'l Symp. on Computer Architecture*, pp. 222-233, June 2005.

[79] C. E. Wu, Y. Hsu, and Y. Liu, "Efficient Stack Simulation for Shared Memory Set-Associative Multiprocessor Caches," *Proc. 1993 Int'l Conf. on Parallel Processing*, Aug. 1993.

[80] Y. Wu and R. Muntz, "Stack Evaluation of Arbitrary Set-Associative Multiprocessor Caches," *IEEE Trans. on Parallel and Distributed Systems*, pp. 930-942, Sep. 1995.

[81] M. Zhang and K. Asanovic, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," *Proc. 32nd Int'l Symp. on Computer Architecture*, pp. 336-345, June 2005.

[82] Z. Zhu and Z. Zhang, "A Performance Comparison of DRAM Memory System Optimizations for SMT Processors," *Proc. 11th Int'l Symp. on High Performance Computer Architecture*, pp. 213- 224, Feb. 2005.

# BIOGRAPHICAL SKETCH

Xudong Shi received his B.E. degree in electrical engineering and M.E. degree in computer science and engineering from Shanghai Jiaotong University in 2000 and in 2003 respectively. Immediately after that, he started to pursue the Doctoral degree in computer engineering at University of Florida. His research interests include micro-architecture design and distributed systems.